An FPGA Implementation of a Power Converter Controller

Tero Kuusijärvi

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 13.10.2017

Thesis supervisor:

Prof. Kari Halonen

Thesis advisors:

D.Sc. (Tech.) Vlad Grigore

M.Sc. Pasi Lauronen

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tero Kuusijärvi

Title: An FPGA Implementation of a Power Converter Controller

Date: 13.10.2017 Language: English Number of pages: 7+84

Department of Electronics and Nanoengineering

Professorship: Electronic Circuit Design

Supervisor: Prof. Kari Halonen

Advisors: D.Sc. (Tech.) Vlad Grigore, M.Sc. Pasi Lauronen

High switching frequencies and control rates in switched-mode power supplies are hard to implement with microcontrollers. Very high clock frequency is required to execute complex control algorithms with high control rate. FPGA chips offer a solution with inherent parallel processing. In this thesis, the feasibility of implementing the control of a typical telecom power converter with FPGA is studied. Requirements for the control system partitioning are considered. The control of a resonant LLC converter is studied in detail and implemented in VHDL. As part of the controller, a high-resolution variable frequency PWM module and floating-point arithmetic modules are implemented. Finally, a complete VHDL simulation model is created and run in different conditions to verify the functionality of the design.

Keywords: LLC, FPGA, high frequency control, DPWM, PFM, power converter

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tero Kuusijärvi

Työn nimi: Tehomuuntajan säädön toteutus FPGA:lla

Päivämäärä: 13.10.2017 Kieli: Englanti Sivumäärä: 7+84

Elektroniikan ja nanotekniikan laitos

Professuuri: Electronic Circuit Design

Työn valvoja ja ohjaaja: Prof. Kari Halonen

Korkeat kytkentä- ja säätötaajuudet hakkuriteholähteissä ovat haastavia toteuttaa mikrokontrollereilla. Monimutkaiset säätöalgoritmit edellyttävät mikrokontrollereilta korkeaa kellotaajuutta. FPGA-teknologia mahdollistaa rinnakkaislaskennan, joka on etu säätösovelluksissa. Tässä työssä tutkitaan FPGA-teknologian soveltumista tyypillisen telecom-tehomuuntajan säätöön. Työssä selvitetään säätöjärjestelmän partitiointia sekä toteutetaan LLC-muuntajan ja sen säätöjärjestelmän simulaatiomalli VHDL-kielellä. Säädön osana toteutetaan korkearesoluutioinen PWM-moduuli sekä liukulukuaritmetiikkamoduuleja.

Avainsanat: LLC, FPGA, DPWM, PFM, tehomuuntaja

Contents

Symbols and abbreviations

Symbols

b_n	bit of index n
C_r	resonant capacitance
D	duty cycle
DR	dynamic range
e_n	exponent bit of index n
F	relative switching frequency
f	frequency
f_{clk}	clock frequency
f_{eff}	effective clock frequency
f_r	first resonant frequency
f_{r2}	second resonant frequency
f_s	switching frequency
$I_{l,rms}$	RMS value of total line (AC input) current
$I_{l1,rms}$	RMS value of line (AC input) current fundamental
I_p	transformer primary current
I_{RMS}	RMS current
I_s	transformer secondary current
L_m	magnetizing inductance
L_r	resonant inductance
m_n	mantissa bit of index n
N_p	number of turns in transformer primary winding
N_s	number of turns in transformer primary winding
p	index of most significant non-zero bit
Q	quality factor
R_{AC}	equivalent load seen from the converter primary side
R_{load}	equivalent load seen from the converter secondary side
s	complex frequency
s_x	sign bit of word x
T_r	isolation transformer
t_{eff}	effective clock period
t_s	switching period
V_{AC}	AC voltage
V_{in}	input voltage
V_L	inductor voltage
$V_{L(av)}$	average inductor voltage
V_{out}	output voltage
V_p	transformer primary voltage
V_{RMS}	RMS voltage
V_s	secondary voltage
θ	angle between RMS voltage and RMS current in linear load
θ_1	angle between line voltage and line current fundamental
Φ_m	magnetizing flux
ω	angular frequency

Abbreviations

AC	alternating current
ADC	analog-to-digital converter
ASIC	application-specific integrated circuit
CCM	continuous-conduction mode
DC	direct current
DPWM	digital pulse-width modulation
DSP	digital signal processor
EMI	electromagnetic interference
FHA	first harmonic approximation
FPGA	field-programmable gate array
GaN	gallium-nitride
HDL	hardware description language
HIL	hardware-in-the-loop
I/O	input/output
IP	intellectual property
LSB	least significant bit
LUT	lookup table
MCU	microcontroller unit
MIPS	million instructions per second
MOSFET	metal-oxide-semiconductor field-effect transistor
MSB	most significant bit
MSPS	million samples per second
PF	power factor
PFC	power factor correction
PFM	pulse-frequency modulation
PID	proportional-integral-derivative
PLL	phase-locked loop
PWM	pulse-width modulation
RAM	random access memory
RMS	root-mean-square
SPI	serial peripheral interface
SRAM	static random access memory
VHDL	very high speed hardware description language
XOR	exclusive or
ZCS	zero-current switching
ZVS	zero-voltage switching

1 Introduction

The requirements for efficiency and flexibility of power converters are ever increasing. Efficient operation over wide operation conditions requires optimization schemes and special operation modes. These requirements demand more advanced control methods. As a result, the industry has turned to digitally controlled high-speed switching converters. Resonant converters are especially attractive since they can be operated in zero-voltage or zero-current-switching modes [1]. There is a drive towards smaller reactive component sizes and higher power densities, requiring ever higher switching frequencies. The maximum switching frequency in a power converter has previously been limited by the switching and conduction losses of silicon-based switching devices. With the emerging of Silicon-Carbide and Gallium-Nitride based low-loss switches, digital control has become the limiting factor for higher switching frequencies.

Digital control of switched-mode power converters is commonly implemented using a microcontroller with DSP capabilities. Special-purpose microcontrollers typically include all the necessary functionalities and peripherals to control a power converter, such as multipliers, timers, analog-to-digital converters and pulse-width modulators, enabling optimization of the total system cost. With matured integrated development environments, designing control algorithms for microcontrollers is a straightforward process. Microcontrollers are also inexpensive. The cheapest models are sold for less than a US dollar.

However, microcontrollers operate in a sequential manner. This means that the maximum instruction throughput is directly related to the system clock frequency. High control frequency requires high performance from the computation hardware itself as well as the peripherals. In one cycle control, there must be enough time in each cycle for the analog-to-digital conversions as well as the control computation. In addition, there must be some processor time reserved for other functionalities such as communication. The number of instructions in a control loop is thus limited, and this sets a limit to the complexity of the control. At the same time, the cost of power supplies is being pushed lower. These factors together make it very difficult to implement a cost-effective real-time control system for switching frequencies approaching 1 MHz with a microcontroller. This is also the reason why field-programmable gate arrays have become more attractive choice for control system implementation.

Unlike microcontrollers, FPGA chips allow for the development of parallel algorithms. The configurable logic interconnections make it possible to calculate parts of the control simultaneously at hardware level. The extra functionality of a control system, such as external communication and system level optimizations can be performed at the same time with the control. These factors remove the control complexity limit, since the control is no longer limited by the system clock frequency. Moreover, several converters can be controlled with one FPGA chip, increasing only the area and cost of the chip but not reducing the available computation time.

In this work, the feasibility of using a field-programmable gate array chip in the control of a typical telecom AC-DC power converter is assessed from cost, performance, modularity and upgradability points of view. Furthermore, an efficient control topology for a PFC-LLC power converter is proposed and an LLC control algorithm capable of 1.2 MHz switching frequency is implemented in VHDL. As part of this implementation, a high-resolution, variable frequency pulse-width modulation architecture is implemented in VHDL. The implementation is verified by HDL simulation. PFC boost converter control implementation is not within the scope of this work.

2 Theory and Background

2.1 Telecom AC-DC Power Converter

In telecommunication sector, the millions of cellular base stations distributed across the globe require high performance DC power supplies. The requirements for power density, smaller size, low harmonic distortion and range of loading conditions are increasing. The purpose of a telecom AC-DC power supply is to convert the line AC voltage of 230 or 110 volts to the DC voltage utilized by the telecom equipment, usually -48 V. The load to be powered may increase and decrease depending on the network utilization at given time. The power converter must be able to keep its output voltage regulated while supplying required amount of current. This power conversion must be achieved with the mentioned requirements taken into account. A popular solution for a switched-mode AC-DC power converter consisting of a boost PFC stage followed by a galvanically isolated DC-DC converter [2] is shown in Figure 1.

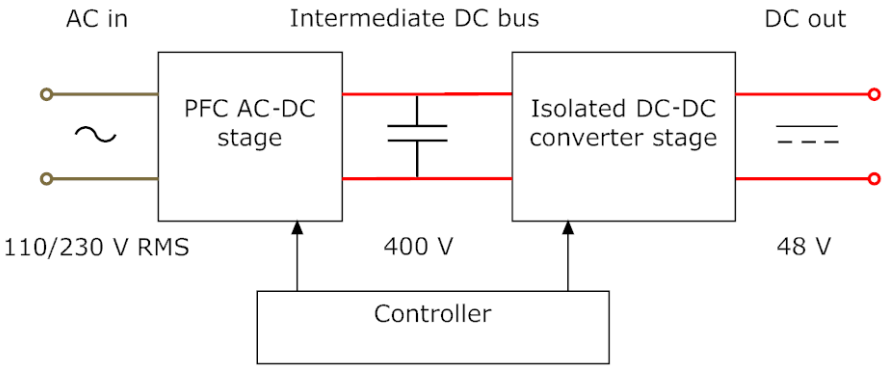

Figure 1: Simplified block diagram of a typical telecom AC-DC power converter

The purpose of the PFC stage (Power Factor Correction) is to keep the input current waveform sinusoidal and in phase with the input voltage. For this stage, a boost PFC topology is commonly employed. A benefit of such PFC stage is the ability to raise the intermediate voltage above the line voltage and also regulate it. This allows using both the 230 V and 110 V AC line inputs with the same design. After the PFC stage, there is a DC-DC stage to add galvanic isolation to the design and to create the 48 V output voltage from the intermediate DC bus voltage. Resonant LLC converter with synchronous rectification is typically used for this stage. The resonant operation allows for soft switching, resulting in highly efficient operation over wide load range. The operation of the two converter stages is governed by the controller block, shown in the bottom. It takes measurements of the various currents and voltages within the converter and generates the appropriate switching sequence for each switching device to keep the operation optimal.

2.2 Power Factor Correction Circuit

The PFC stage is essentially a rectifier coupled with a boost converter: its purpose is to shape the line current drawn by the power converter to sinusoidal form and keep it in phase with the input AC voltage while regulating the intermediate DC bus voltage. The sinusoidal waveform of line current minimizes its distortion. The distortion of the line current is linked to the efficiency of the whole power converter. High distortion equals high harmonic content which in turn means high reactive power and thus higher losses. Furthermore, distorted current might also distort the grid AC voltage, and that is why it should be minimized. Standards such as IEC61000-3-2 [3] regulate the maximum allowed levels of distortion in devices that are connected to the electric grid.

2.2.1 Power Factor

Power factor is defined as the relation of real power to apparent power in a system [4, p. 524]. It represents the efficiency of the use of real power within the system. Power factor for linear loads is given by:

$$PF = \frac{V_{RMS} I_{RMS} \cdot cos(\theta)}{V_{RMS} I_{RMS}} = cos(\theta), \tag{1}$$

where θ is the angle between the RMS line voltage and RMS line current. Since the real power utilized by the system is the instantaneous product of voltage and current, high RMS values of voltage and current do not correspond to high power delivered to load, if the power factor is low. However, high RMS current does contribute to higher conductive losses, even if it is not in phase with the line voltage.

Switched-mode power supplies are not linear power systems. Rectifier bridges draw distorted currents, which degrade the power factor considerably. This is why it is not enough to keep the line current in phase with the input voltage. The general equation for the power factor is given by [4, p. 525]:

$$PF = \frac{I_{l1,rms}}{I_{l,rms}} \cdot cos(\theta_1), \tag{2}$$

where $I_{l1,rms}$ is the rms value of the fundamental component of the line current, $I_{l,rms}$ is the rms value of the line current, and θ_1 the angle between the line voltage (assumed sinusoidal) and the fundamental component of the line current. Minimizing distortion implies minimizing the harmonic components of the line current to make the total rms value of line current equal to the rms value of the fundamental component. The goal is to shape the line current as close to purely sinusoidal as possible. There are passive and active means to improve the power factor. In this work, the active boost PFC topology shown in Figure 2 is used. With a boost PFC it is possible to achieve power factors higher than 98%.

2.2.2 Structure and Operation

The PFC boost converter in Figure 2 consists of the input rectifier stage, boost inductor L, switch Q, boost diode D and the intermediate DC bus capacitor [4, p.

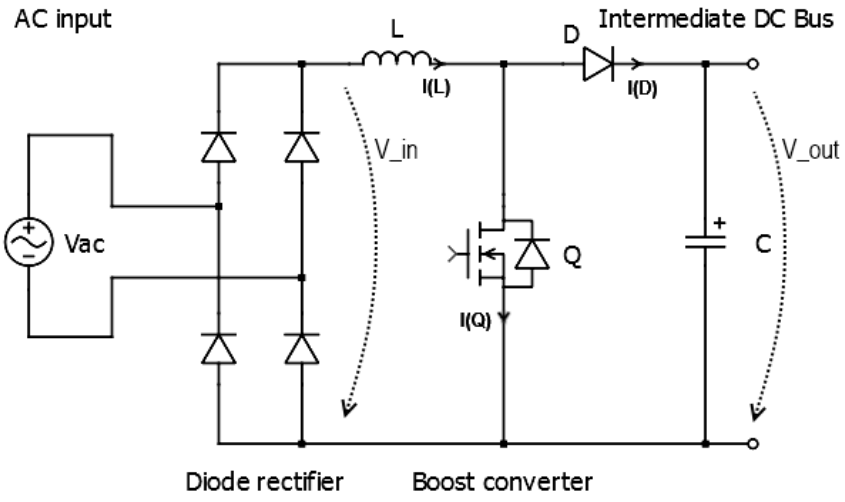

Figure 2: The PFC boost converter with a diode rectifier

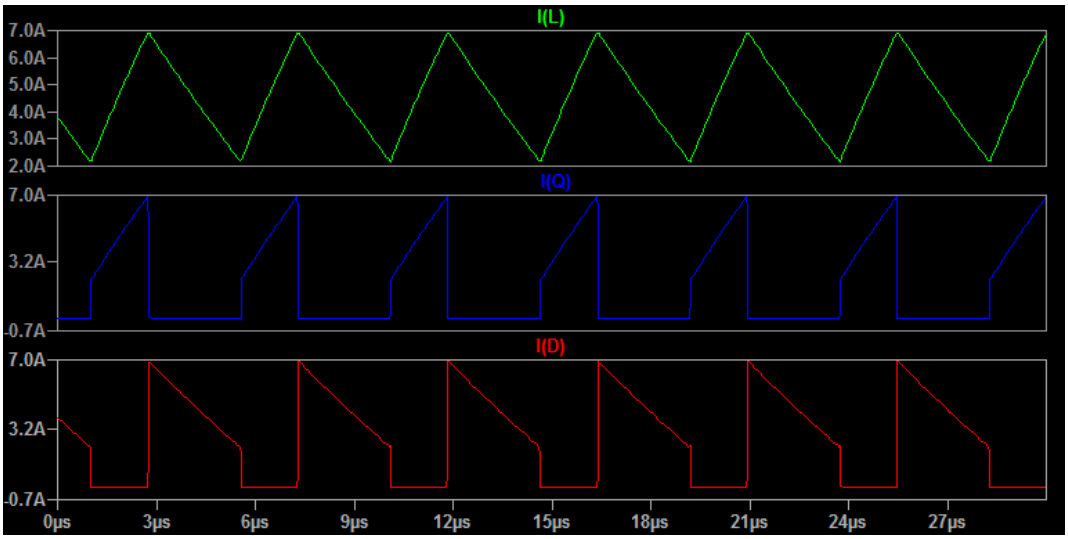

Figure 3: Current waveforms of the boost converter

528]. The boost inductor is located in the path of the input current, while the switch is connected to the boost inductor and the negative terminal of the intermediate DC bus capacitor. This configuration has advantages over buck and buck-boost type converters. When the converter operates in continuous-conduction mode (CCM), the input current is always greater than zero. It is also continuous, since it is not disrupted by the switch. The current stress of the switch is thus lower in a boost type converter. Additionally, the boost inductor smooths the input current, resulting in lower EMI generation [4, p. 529].

When the switch Q conducts, the input current flows through the boost inductor and the switch. The voltage over the inductor L is equal to the input voltage, which

in this case is the rectified line voltage. The current through the inductor increases, while the load, which in this case is the LLC converter, is supplied with current from the discharging intermediate DC bus capacitor. Since the inductor corresponds to a short circuit for low frequencies, there cannot be an average non-zero voltage over the inductor. When the switch does not conduct, the input current flows through the boost inductor to the load and the intermediate DC bus capacitor. In this case, the voltage over the inductor is equal to the difference of input and intermediate DC bus voltages:

$$V_L = V_{in} - V_{out}. \tag{3}$$

Thus with a duty cycle D, the average inductor voltage is:

$$V_{L(av)} = 0 = D \cdot V_{in} + (1 - D) \cdot (V_{in} - V_{out}) \tag{4}$$

$$(1 - D) \cdot V_{out} = D \cdot V_{in} + (1 - D) \cdot V_{in} \tag{5}$$

$$V_{out} = \frac{1}{1 - D} V_{in}. \tag{6}$$

As can be seen in Equation 6, the output voltage of a boost converter depends directly on the duty cycle. It is thus easy to control a boost converter, since increasing load or decreasing input voltage can both be compensated by simply increasing the duty cycle towards 1. However, in a boost PFC this is not enough. The intermediate DC bus voltage must be regulated while simultaneously keeping the line current waveform sinusoidal. This can be accomplished by appropriate control. Since the value of duty cycle corresponds to the period of time the current of the boost inductor increases, the line current can be shaped with the duty cycle. For example, when the line voltage wave reaches its peak value as its derivative is close to zero, duty cycle of 50% keeps the inductor current and equivalently the input current approximately constant. After this, the average duty cycle is decreased in phase with the sinusoidal line voltage. It is important to note here that the switching frequency is much higher than the line frequency. If the duty cycle is varied synchronously with the line voltage, the average inductor current is approximately sinusoidal within a line cycle. Varying the duty cycle results in ripple effect on the intermediate DC bus voltage.

To decrease the losses of the PFC stage, it is possible to use bridgeless design for the input rectifier. Several such implementations have been proposed [5]. The advantage of this is that some of the diodes within the path of the input current can be removed, and thus the conduction losses reduced. However, bridgeless designs require more complicated control because of greater number of controlled switches. Furthermore, bridgeless design introduces other challenges like increased common-mode noise [5].

Another technique to reduce the conduction losses of the PFC stage is to use multiple interleaved boost converters [6]. The conduction losses, directly proportional to the square of the current, are reduced to a quarter by dividing the current equally between two interleaved converters. Another benefit of this is the ability to use smaller inductors, because the current ripple in each interleaved path sums up with a 180 degree phase shift, compensating the total ripple. From the grid side, this

seems like doubled switching frequency, which naturally reduces ripple. Additionally, this reduces the line current distortion and transient overshoots [7]. Interleaved PFC is best controlled digitally because of the need to keep the interleaved paths 180 degrees out of phase.

2.3 LLC Resonant Converter

Conventional PWM power converters operate using hard switching. This means switching where the switch device has to endure high voltage or current during the switching. Hard switching causes great stress to the power electronic components and also losses in the form of heat. The losses originate from the fact that during the switching, there is both current running through the component as well as a voltage over it. Moreover, rapidly changing currents and voltages cause increased electromagnetic interference. To overcome these problems, research efforts were focused to resonant converter topologies such as LLC converter [4, p. 410]. The general idea was to use resonant tanks to create sinusoidal voltage and current waveforms, allowing the switching to be timed to the zero-crossing points. From there come the terms zero-voltage switching (ZVS) and zero-current switching (ZCS). A characteristic of LLC converter is that it is controlled with switching frequency while the duty cycle is fixed to approximately 50%, whereas the conventional PWM converters are controlled with duty cycle while keeping the switching frequency constant.

2.3.1 Structure

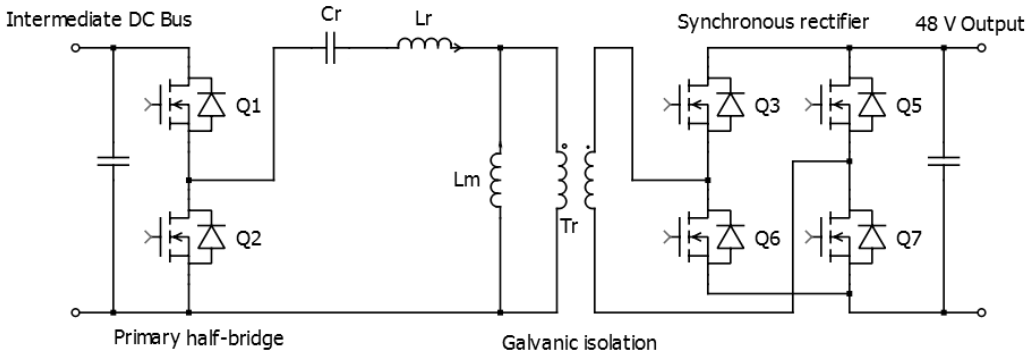

Figure 4: Circuit diagram of synchronously rectified LLC converter

There are several possibilities to arrange the reactive components of the resonant tank that result to different characteristics. One of those possibilities is the LLC converter [8]. In LLC converter the resonant tank is made of three reactive components, two inductors and a capacitor. LLC converter has advantages over the other configurations, namely efficient operation over wide load conditions and wider gain variation with narrower frequency control range [8]. Additionally, in LLC

configuration it is possible to utilize the parasitic inductances of the transformer by integrating all the inductive components into the transformer. This is very beneficial from the system integration point of view. Common to all configurations is the basic operation principle: a square wave is applied to the resonant circuit by the primary bridge circuit. The bridge circuit can be of either half-bridge of full-bridge type. In this work, the half-bridge topology is used. The resonant circuit filters the applied wave, and approximately sinusoidal current flows through the windings of the isolation transformer. The current is rectified in the secondary side of the transformer, resulting in DC-DC conversion.

The secondary side rectifier could be realized with diodes. However, with passive rectification there exists a constant voltage drop over the rectification diodes. With low voltages and large currents, the resulting power loss grows too high. Synchronous rectification is a viable solution. Replacing the diodes with switches, such as the MOSFETs in Figure 4, the conduction losses can be almost eliminated. Synchronous rectification requires that the switching takes place when the current through the switch is approximately zero, in order not to introduce additional losses originating from the switching. To achieve this the phase difference between the primary switching signals and the secondary switching signals must be calculated.

The three reactive components of the resonant tank cause the converter to have two resonant frequencies, given by [9, p. 7/64]:

$$f_r = \frac{1}{2\pi\sqrt{L_r C_r}} \tag{7}$$

$$f_{r2} = \frac{1}{2\pi\sqrt{(L_r + L_m)C_r}}, \tag{8}$$

where L_m, L_r and C_r are the magnetizing inductor, resonant inductor and resonant capacitor respectively. The primary resonant frequency in Equation 7 is the frequency that is commonly referred to as the resonant frequency. The second resonant frequency f_{r2} is at lower frequency than the primary resonant frequency. The impedance of the resonant tank can be either inductive or capacitive, depending on the switching frequency. When the LLC converter is driven below the lower resonant frequency, the input impedance of the resonant tank is capacitive. The input impedance is inductive when the resonant tank is driven above the higher resonant frequency [9, p. 7/64]. Between the resonant frequencies, the input impedance depends on the load condition, and can be either inductive or capacitive.

2.3.2 Resonant Tank Gain

The analysis of resonant switching converters is thought to be more complex than in the case of conventional PWM converters. To address this, it is assumed from here on that from the square wave input only the first fundamental harmonic component contributes to the transfer of electrical power and the rest of the harmonic content is neglected. This greatly simplifies the analysis of the converter, and allows the usage of classical AC circuit analysis to study the dynamics of the system. This is called

first harmonic approximation (FHA). In addition to the LLC gain characteristics, it is used in the control algorithm of the synchronous rectifier to find the optimal switching instants for the secondary side switches.

Simplified circuit consisting of the primary resonant tank, magnetizing inductance, transformer and the load is shown in Figure 5. The synchronous rectifier is combined

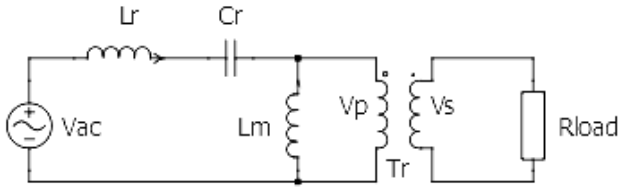

Figure 5: Simplified diagram of resonant tank, transformer and load.

with the actual load to a single load R_{load}, seen as an AC load from the transformer. The voltages from the dotted terminals to the undotted terminals for primary and secondary sides of the transformer are respectively the following [10, p. 137]:

$$V_p = N_p \frac{d\Phi_m}{dt} \tag{9}$$

$$V_s = N_s \frac{d\Phi_m}{dt}, \tag{10}$$

where $\mathrm{N_p}$ and $\mathrm{N_s}$ is the number of turns on either side and Φ_m is the magnetizing flux of the transformer. An ideal transformer is assumed here, and for that reason no leakage fluxes or losses are included in this analysis. Dividing Equation 9 by Equation 10 yields:

$$\frac{V_p}{V_s} = \frac{N_p}{N_s}. \tag{11}$$

Conservation of energy means that the input power and the output power of the transformer must be equal. This means that:

$$V_p I_p = V_s I_s \tag{12}$$

$$\rightarrow \frac{V_p}{V_s} = \frac{I_s}{I_p} = \frac{N_p}{N_s}. \tag{13}$$

The relationship between the primary current and the secondary current is then:

$$I_s = I_p \frac{N_p}{N_s}. \tag{14}$$

From Equation 14 it can be seen that the secondary current is simply the primary current scaled by the turns ratio constant. This means that the primary and secondary currents are in phase, and the circuit in Figure 5 can be further simplified by including the transformer and the load to a single AC load, R_{AC}. The simplified circuit is shown in Figure 6.

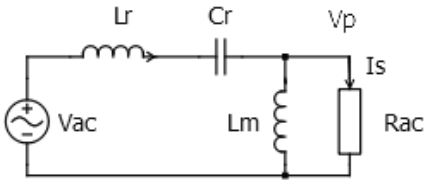

Figure 6: Further simplified resonant circuit.

Using Laplace transforms to obtain the complex impedances, we get for the complex primary voltage:

$$V_p = \frac{\frac{sL_mR_{AC}}{sL_m+R_{AC}}}{\frac{sL_mR_{AC}}{sL_m+R_{AC}} + sL_r + \frac{1}{sC_r}} V_{AC} \tag{15}$$

$$V_p = \frac{sL_mR_{AC}}{sL_mR_{AC} + s^2L_mL_r + sL_rR_{AC} + \frac{sL_m+R_{AC}}{sC_r}} V_{AC} \tag{16}$$

$$V_p = \frac{s^2C_rL_mR_{AC}}{s^2C_rL_mR_{AC} + s^3C_rL_mL_r + s^2C_rL_rR_{AC} + sL_m + R_{AC}} V_{AC} \tag{17}$$

$$V_p = \frac{s^2C_rL_mR_{AC}}{s^3C_rL_mL_r + s^2C_rR_{AC}(L_m + Lr) + sL_m + R_{AC}} V_{AC}. \tag{18}$$

Substituting jω in place of the complex frequency s yields:

$$V_p = \frac{-\omega^2C_rL_mR_{AC}}{-j\omega^3C_rL_mL_r - \omega^2C_rR_{AC}(L_m + Lr) + j\omega L_m + R_{AC}} V_{AC} \tag{19}$$

$$V_p = \frac{-\omega^2C_rL_mR_{AC}}{j(\omega L_m - \omega^3C_rL_mL_r) - \omega^2C_rR_{AC}(L_m + Lr) + R_{AC}} V_{AC}. \tag{20}$$

Substituting switching frequency $2\pi f_S$ in place of ω yields:

$$V_p = \frac{-(2\pi f_s)^2C_rL_mR_{AC}}{j((2\pi f_s)L_m(1 - (2\pi f_s)^2C_rL_r)) - (2\pi f_s)^2C_rR_{AC}(L_m + Lr) + R_{AC}} V_{AC}. \tag{21}$$

Quality factor Q and relative switching frequency F are the following:

$$Q = \frac{\sqrt{\frac{L_r}{C_r}}}{R_{AC}} \tag{22}$$

$$F = \frac{f_s}{f_r}. \tag{23}$$

Using quality factor and relative switching frequency instead of R_{AC} and f_s in Equation 21 results to:

$$V_p = \frac{-(2\pi F f_r)^2 C_r L_m \frac{\sqrt{\frac{L_r}{C_r}}}{Q}}{j((2\pi F f_r)L_m(1-(2\pi F f_r)^2 C_r L_r)) - (2\pi F f_r)^2 C_r \frac{\sqrt{\frac{L_r}{C_r}}}{Q}(L_m + Lr) + \frac{\sqrt{\frac{L_r}{C_r}}}{Q}} V_{AC}. \tag{24}$$

Using Equation 7, this simplifies into:

$$V_p = \frac{-F^2 L_m}{j \cdot Q L_m F(1-F^2) - F^2(L_m + L_r) + L_r} V_{AC} \tag{25}$$

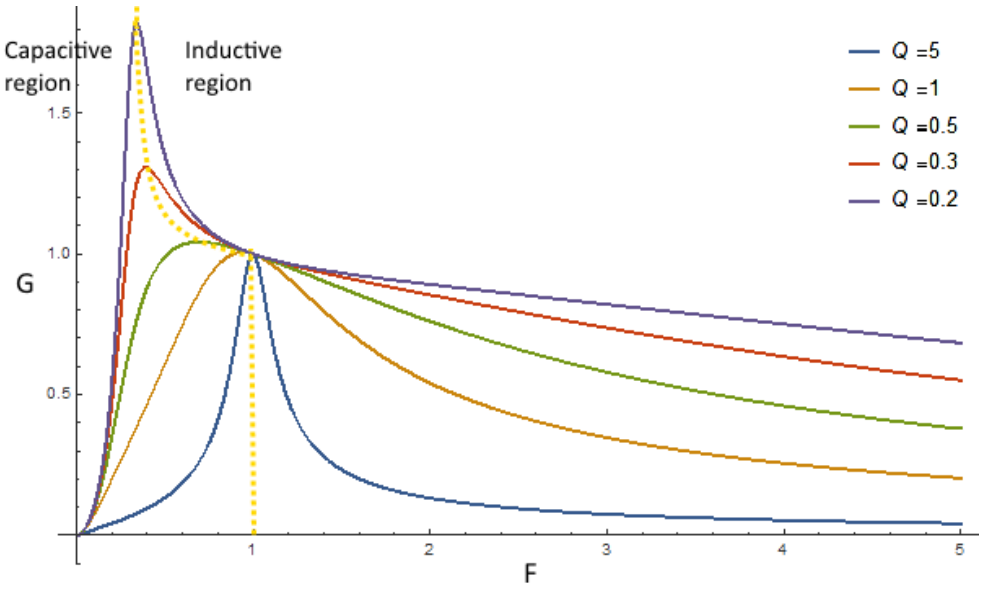

Figure 7: LLC resonant tank gain as function of relative switching frequency

The equation for the resonant tank gain *G(Q, F)* is then:

$$G(Q,F) = |\frac{V_p}{V_{AC}}| = \frac{F^2 L_m}{\sqrt{(Q L_m F(1-F^2))^2 + (L_r - F^2(L_m + L_r))^2}}. \tag{26}$$

As can be seen from Figure 7, the gain of the resonant tank depends on the switching frequency of the primary side bridge circuit. The LLC converter is controlled by adjusting the switching frequency near the resonant point. Zero-voltage switching requires that the resonant tank impedance is inductive [9, p. 15/64]. The switching frequency must then be limited to the inductive frequencies. Inductive impedance increases with frequency, which is why the gain of the resonant tank is lower at higher frequencies. The controller must respond to decreasing input voltage or increasing load by reducing the switching frequency. With heavy loads (corresponding to quality factor of 5 in Figure 7), the lowest allowed frequency is the resonant frequency. With lighter loads, the switching frequency can be even lower, as long as it is in the inductive region. All the gain curves coincide at the resonant frequency, where it has value of unity. This is the operation point when the load condition is at the specified nominal level. As will be shown next, it is the point of most efficient operation.

2.3.3 Operation

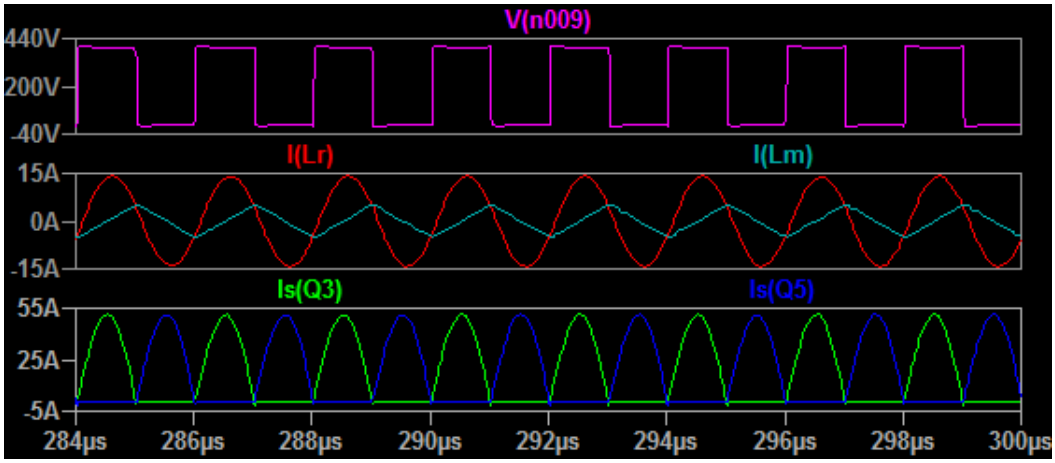

Figure 8: LLC waveforms at resonant frequency

in Figures 8 - 10, the voltage and current waveforms of the LLC converter are plotted. *V(n009)* refers to the voltage applied to the resonant tank by the primary switches. *I(Lr)*, *I(Lm)*, *Is(Q3)*, and *Is(Q5)* are the resonant inductor current, magnetizing current, and source currents through secondary switches *Q3* and *Q5* (in Figure 4) respectively. In Figure 8 the circuit is operated at the resonant frequency. The resonant inductor current *I(Lr)* is approximately sinusoidal and the magnetizing current *I(Lm)* is triangular. The difference of these two currents flows through the windings of the transformer. On the secondary side, the current flows through the rectifier switches, which can be operated in ZCS conditions. On the primary side, ZVS conditions are achieved in turning on the switches. During turn-off, there is always some current flowing through the switches, causing turn-off losses. It should be noted here that the resonant inductor current is not at its peak during the switching, but rather has reversed its direction of change and is approaching zero. At the points

when the resonant inductor current meets the magnetizing current, the conducting primary side switch is turned off. The inductive operation region guarantees that the current will keep flowing. The only path for the current is through the body diode of the other switch. To force the current through the body diode, the voltage at the switching node is forced to increase to forward bias the body diode. After this has happened, the voltage over the switch is just the forward voltage of the body diode, allowing ZVS at the turn-on.

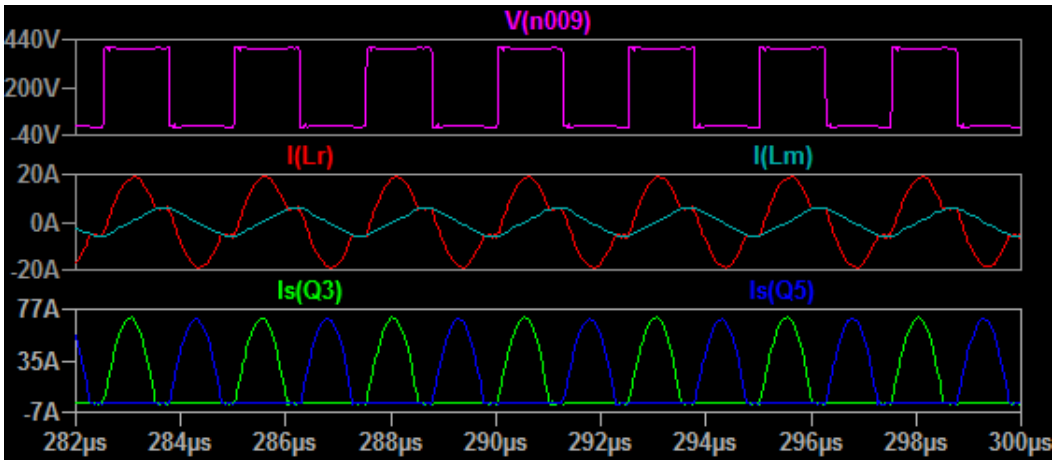

Figure 9: LLC waveforms below resonant frequency

in Figure 9, the converter is operated below resonant frequency. Now the square voltage period is longer than the resonant period of the resonant tank. The resonant inductor current *I(Lr)* meets the magnetizing current *I(Lm)* before the end of the switching half-cycle. After this point, no current flows through the primary windings of the transformer until the beginning of the next switching half-cycle. The current is simply circulated in the primary side, resulting in increased conduction losses. On the secondary side, the duty cycle of the switches must be reduced, since the duration of the current pulses are now only a portion of the primary switching cycle.

in Figure 10, the converter is operated above resonant frequency. Now the square voltage pulse is too short to allow for the resonant inductor current to complete the resonant cycle. On the primary side, this increases the turn-off losses of the switches, as the resonant inductor current is closer to its peak during the switching. On the secondary side, ZCS is no longer possible. As can be seen in the figure, the secondary current pulses are disrupted and the new half-cycle is started, resulting in inreased switching losses.

As a whole, LLC control is relatively complicated. The circuit gain is controlled with varying frequency rather than duty cycle. The control response must be different for different load conditions, as can be seen in Figure 7. The secondary duty cycle is not necessarily the same as the primary duty cycle. This is demonstrated in Figure 9. In the same figure, it can be seen that in addition to calculating the duty cycle, the switch control pulses must be slightly advanced in relation to the primary pulses to ensure zero-current switching. This advancing or delaying depends on the

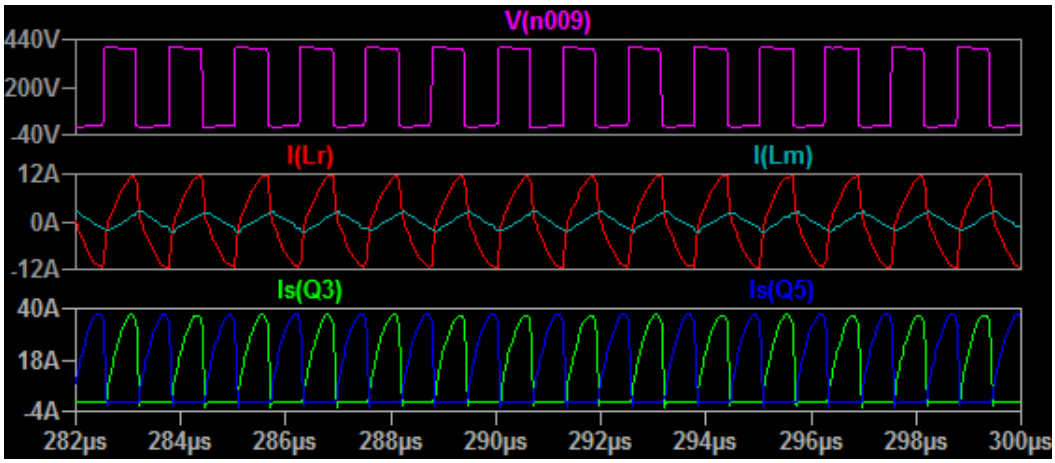

Figure 10: LLC waveforms above resonant frequency

primary switching frequency. This synchronization is somewhat complicated, and it is further analyzed in section 3.4. Tight regulation of the output voltage typically requires a two-stage nested control, with output voltage and primary current being the measured quantities that are then compared to their reference values [11]. As the goal is to move to higher switching frequencies in the future to increase power density and reduce component sizes, it requires a high performance controller to run a complex control algorithm. This is why the FPGA technology is becoming more and more interesting for control implementations.

2.4 Field-Programmable Gate Array Technology

Field-programmable gate-array is essentially programmable logic. FPGA chips make it possible for the developer to implement logic functions in hardware. This allows implementing extremely fast algorithms and parallel processing without overhead. FPGAs can be used for most of the applications that application-specific integrated circuits (ASIC), but unlike ASICs, their logic functionality is not fixed during production but rather defined later by the developer. For this reason, using an FPGA chip instead of an ASIC is an attractive option in prototyping, low volume production and situations where it is necessary to be able to modify and update the logic within the final product on the field. While having reduced time-to-market, FPGAs are not fully customizable like ASICs and therefore cannot be as precisely designed as ASICs. An ASIC developer can draw wires in the design as desired, while FPGA designer must rely on pre-existing wiring and connections.

2.4.1 Structure

FPGA chip consists of two basic building blocks, logic and interconnections [12, p. 3–4]. In addition, there are number of I/O pins to connect the chip to the rest of the printed circuit board. Together these blocks form the logic fabric of the FPGA. The

logic resources are commonly realized with N-input lookup tables (LUT). LUTs are basically truth tables, which can implement any N-input combinational logic function. Lookup tables are programmed with a truth table of the implemented function. A 4-input LUT is equal to a 16-bit read only memory connected to a multiplexer. Complex combinational logic functions can be implemented with multiple lookup tables interconnected in a specific manner.

In addition to the lookup table, the other basic element in a logic slice is the register, or a flip-flop. The register is essentially a 1-bit memory element, which samples the input at a certain point of the provided clock signal and transfers the input to its output. Without memory, it would be impossible to implement state machines with an FPGA, and only combinational logic would be possible. In addition to memory, registers allow the implementation of delays and pipelines. Registers are also the root of the timing considerations of an FPGA design. To avoid metastable states and ensure correct operation, the input must be stable for a certain period of time before and after the sampling moment. These periods are called setup and hold times respectively. Using too high clock frequency for a given logic path would result in too little time for the settling of the input data of the register at the end of the path, causing its output data to be invalid at the next sampling moment. The timing requirements can be relaxed by using a lower frequency clock or inserting registers within the logic path.

Besides the LUT and the register, there is usually a multiplexer within each logic slice. With the multiplexer, output of either the LUT or the register can be chosen as the output of the block. The multiplexer is also the element that allows flexible interconnection of sequential and combinational logic.

There are usually ASIC sections, called hard intellectual property cores (IP), designed within an FPGA to allow for more flexible clock management and fast, resource efficient arithmetic operations. These blocks include phase-locked loops, delay lines, memory blocks and multiply-accumulators. These blocks can be conveniently connected to the logic fabric to improve the performance of related logic or preserve the logic resources of the chip.

2.4.2 Routing, Interconnect and Clock Distribution

In a single FPGA chip, there can be thousands of individual logic slices. To map a complex logic function on the chip, the slices must be interconnected in a meaningful way. The commonly used routing architecture is the island-style architecture. In this architecture, the logic slices are arranged in a two-dimensional array with interconnections and routing in between. At the intersections of routing channels, there are switchboxes to connect the logic slices and the wiring itself in a configurable manner.

The clock signal as well as a global reset signal usually have dedicated routings, inputs and buffers to minimize skew and ensure an even distribution to the entire FPGA chip. There can be several clock domains with different frequencies on a single FPGA. Often, some of the logic running on the chip has tighter timing requirements than the rest. This allows using the minimum required clock frequency within a

certain clock domain in order to save dynamic power. Care must be taken when routing signals from one clock domain to another. Since the differing clocks can be of different frequency and phase, it is possible that the register in the target domain samples the input data during a transition within the source clock domain, resulting in corrupted data. This can be avoided by using multiple buffer registers between clock domains to reduce the likelihood of unsettled input data.

2.4.3 Design Flow

The FPGA design flow is similar in the development environments of the major manufacturers. The steps are [13]:

1. HDL description
2. Translation and synthesis
3. Design implementation
4. Physical implementation

In the first step, a HDL description is written. Popular hardware description languages are VHDL and Verilog, derived from Ada and C respectively [12, p. 129]. At this point, the design is created as several modules, which can usually be individually simulated to ensure their functionality before interconnecting them as a larger design. Care must be taken to write the kind of description, which corresponds to the available hardware elements in some way. It is possible to write HDL code that is not synthesizable. Simulation after and during this step is called pre-synthesis simulation.

In the second step, the written description is synthesized. This means creating a netlist out of the design [13]. The netlist describes the logic gates needed to implement the design as well as the way those components are connected. Simulation after this step is call post-synthesis simulation.

In the third step, the netlist is mapped to the available FPGA resources and subsequently placed and routed on the target FPGA. The best physical locations for the functional blocks are found and routing between the blocks is calculated. The result of this step is the configuration data that can be programmed on to the FPGA chip. Simulation after this step is called post place-and-route simulation.

The data that specifies the configuration is called the configuration bitstream [12, p. 402]. In the final step, the bitstream is actually programmed into the target FPGA chip. After this step, the functionality of the design can be verified for example with a logic analyzer or a hardware-in-the-loop (HIL) simulation. As mentioned in the previous subsections, there are many elements in an FPGA that can be configured. These elements are the truth tables of LUTs, initial values of the D flip-flops, the select value of the multiplexers of the logic slices and the configurations of the switchboxes. Both the logic and the interconnect configuration bits are commonly static SRAM. The bitstream is usually stored in a non-volatile flash memory, external or internal. From there, the configuration data is loaded to the FPGA's internal SRAM memory

at the time of power-on start-up or during a manual reset at run time. SRAM is a type of volatile memory, which keeps its value only when it is powered [12, p. 17].

FPGA development allows for code reuse just like in software engineering. Because of the relatively large workload involved in creating a complex digital design from scratch, it is common to use previously designed and verified modules to speed up the development cycle. The major FPGA vendors have reusable modules, called soft IP cores, available for purchase and some for free.

2.5 Digital Control of PFC-LLC Power Converter

The main elements of digital switched mode power supply control system are [14, p. 6]:

1. Data acquisition
2. Control law generation
3. Digital PWM generation

2.5.1 Data acquisition

As digital controllers operate with sampled and quantized measurements, analog-to-digital converters are needed to get information about the controlled process. These measurements are essentially the feedback loop of the control system. In the case of PFC-LLC power converter, we need several measurements for the control. These include:

1. AC input voltage
2. PFC phase current(s)
3. PFC output or intermediate voltage
4. LLC primary current
5. LLC output current
6. LLC output voltage

Typical ADCs have an input range of [0, 3.3] volts. However, many of the measured quantities have too large values to be directly measured with an ADC. It is necessary to scale those values down to the suitable range. Additionally, some measurements must be obtained across the isolation barrier. Different quantities require different sensing methods. For DC voltages, the scaling is done with a voltage divider. This is simply a chain of resistors in series, which scales the high voltages to lower voltages. While a resistor chain is a simple and low-cost measurement method, it does not provide isolation. A surge voltage in the input of the voltage divider can propagate past the voltage divider, possibly destroying some components. To

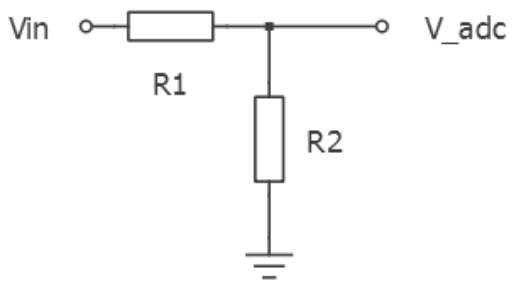

Figure 11: A voltage divider

make sure that the resistors themselves will not be destroyed by surges, they must be properly selected. The voltage divider is shown in Figure 11. The equation for the scaled-down voltage V_adc is:

$$V\_adc = Vin \cdot \frac{R2}{R1 + R2}. \tag{27}$$

For DC currents, the most relevant options for obtaining the measurement and scaling the value are shunt resistors and Hall-effect sensors [15, p. 133]. Shunt resistor is a resistor placed in the path of the current to be measured. As the current flows through the resistor, a detectable voltage drop is formed over the resistor. The measurement scale can be adjusted by sizing the resistor properly or using an amplifier circuit. However, the resistor should be of small value, since it affects the surrounding circuit. In addition to the voltage drop over the resistor, some power is inevidently dissipated within the resistor. It should be noted that measuring current with a shunt resistor provides no isolation. The shunt resistor is shown in Figure 12. The equation for the voltage drop is:

$$V\_adc = I \cdot R\_shunt. \tag{28}$$

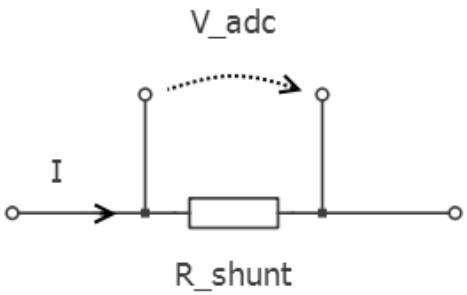

Figure 12: A shunt resistor

Instead of measuring the voltage drop in a resistor, Hall-effect sensors measure the magnetic field created by the current [15, p. 133]. This has the advantages

of not affecting the measured circuit and providing electrical isolation. Hall-effect sensors are useful for measuring large currents, since those generate larger magnetic fields, leading to smaller relative error. With small currents however, factors such as the orientation of the sensor in relation to earth's magnetic field can affect the measurement considerably. Hall-effect sensors are also very sensitive to differences in distance to the measured wire as well as the wire geometrics [15, p. 134].

For AC quantities, a straightforward sensing method is to use a transformer. With proper turns ratio, the secondary winding of a voltage transformer provides a voltage which can be directly measured, as is shown in Equation 11. In case of AC current, a current is obtained from the secondary winding. If a resistor is connected to the secondary winding, voltage directly proportional to the primary current can be measured over the resistor. With proper sizing of the transformer and the resistor, negligible additional load is presented to the primary side. As an added benefit, transformers provide isolation.

The requirements for the sampling speed and bit resolution of the measurements vary, but generally the measurements that are needed within the control loops have the strictest requirements. The requirements depend on the control loop frequency as well as the dynamic and static requirements of the control. The control loop frequency means the rate at which the control algorithm is executed. The ADC's sampling speed must be at least equal to the control loop frequency, and in practice higher than that. In one-cycle control, it is necessary to have ADC sampling rate higher than the control loop frequency to leave a portion of the control period for the control law generation.

Dynamic requirements refer to the way the controlled system responds to changes in its operating conditions. Higher resolution of the ADC means that the output word of the ADC is wider in bits. This means that the measured quantity is represented by a larger integer value than with a smaller resolution ADC. When the measured quantities change their value, that change is more easily detected with a higher resolution ADC. Furthermore, when the measured quantity differs from the set reference point, the error quantity seen by the controller has a greater value. Finer precision also helps the control system to achieve more stable operation around the set reference. This requirement is usually defined as the maximum allowed ripple in a given quantity.

2.5.2 Control law generation

The output of the controlled process, in this case currents and voltages, depend on both past and present inputs. The past inputs have caused the process to enter a certain state: its node voltages and branch currents have a certain value at a given time instant. Based on this state, certain actions must be taken to direct the process towards the desired state. As the process here is a power converter, this generally means taking appropriate actions to reach a desired output voltage. Control law means the function of process states, which determines the actions to be taken. In an LLC converter, at some instant, the output voltage might be less than the reference voltage. The control law dictates that in this situation, the action to be taken is

to reduce the switching frequency of the primary side switches in order to increase the gain of the resonant tank and subsequently the output voltage. Similarly, the control law dictates that an output voltage exceeding the reference voltage should be compensated by increasing the switching frequency.

Control law also determines the magnitude of the response to a certain magnitude of error between the reference and the measured quantity. This is done using proportional-integral-derivative (PID) control structure. PID control is probably the most used control structure within industrial control. It is very robust, since it doesn't require information about the process itself. The process output is measured and subtracted from the reference to yield an error term. A control command is calculated as a sum of three terms: one directly proportional to the instantaneous error term (P), one proportional to the change, or derivative of the error term (D) and one proportional to the accumulated or integral of the error term (I). In digital control, it is sometimes good practice to use only P and I terms, since there is always noise present in measurements. From the control point of view, noise equals to abrupt changes, causing strong responses because of the derivative term. This kind of action might actually worsen the dynamic properties of the control and possibly make it unstable. This is the reason why only PI control is used in this work.

The control law is generated by a digital signal processor, a microcontroller or an FPGA chip. The FPGA or microcontroller is called a controller. The controller takes as its inputs the necessary measurements, and from them calculates the digital command word, which represents the appropriate response to a given process state at a given instant. This digital command word is then fed to an actuator, in this case a digital pulse-width-modulation unit (DPWM), which is the means by which the controller can affect the controlled process.

2.5.3 Digital PWM generation

The DPWM can be thought of as a digital-to-analog converter: it converts the digital command word to an analog signal, which has the essential information modulated in the relative width of its pulses as well as in its harmonic content. When the frequency of the modulated signal is controlled, the module is called pulse-frequency-modulator (PFM). In this work, both PWM and PFM are used. The relative width of the pulses is called the duty cycle, which is usually given as a percentage. A 0% duty cycle corresponds to a signal that is constantly zero and duty cycle of 100% corresponds to a constantly high signal. A duty cycle of 50% is equal to a square wave: the signal is at a logic high for as long period of time as it is at the logic low. This series of pulses is used to control the state of the semiconductor switches within the power converter. For example, in the LLC primary half-bridge there are two switching devices that are operated out of phase, allowing a square wave of about 400 V to be applied to the resonant tank.

Unlike analog PWM modules, the PWM output of a digital modulator is not continuous but rather has a finite number of possible output duty cycles. The width of the digital command word determines the resolution of the DPWM module. Since the command word represents a value between 0% and 100%, or [0, 1], the number

of possible values of the command word relates to the maximum precision with which the desired duty cycle percentage can be obtained. For example, an 8-bit DPWM module has 256 possibilities for the duty cycle, which means that the finest possible adjustment of the duty cycle is about 0.39 percentage points. The frequency resolution is limited by the period of time that the least significant bit (LSB) of the frequency command word corresponds to. If an LSB of the frequency command word is equal to 1 ns, and the operation frequency is initially 1 MHz, then increasing the frequency command by one LSB results in a period of:

$$t_s = 1\mu s + 1ns = 1.001\mu s \tag{29}$$

$$f_s = \frac{1}{t_s} \approx 999kHz. \tag{30}$$

The frequency resolution is then 1 kHz around the frequency of 1 MHz, or about 10 bits:

$$2^{10} = 1024 \approx \frac{1MHz}{1kHz}. \tag{31}$$

As can be seen from above, the frequency resolution in bits depends on the frequency around which the control system operates. The higher the control frequency, the lower is the available PWM frequency resolution in bits. The resolution is important from control point of view, since the accuracy of the control is limited by the PWM resolution. The PWM resolution must be at least higher than the ADC resolution to avoid limit cycling [16].

There are a couple of possible DPWM architectures for FPGAs [17] and several implementations that are based on these architectures or hybrids of them [17]-[21]. These architectures are the counter based and delay line based architectures. The counter based architecture is in principle similar to the analog PWM: The output is obtained by comparing the PWM command word to a carrier signal. In DPWM, the carrier signal is a digital counter, and the comparator is a digital comparator. The counter starts from zero, with the PWM output at logic high. The counter then increases its value at every clock cycle, until the counter value is equal to the digital command word. This is where the PWM output is reset to logic low, and is kept in that value until the counter increases to its maximum value and is set to zero again. This is a simple architecture and easy to implement with FPGA resources. The disadvantage of this architecture is that the resolution of the PWM signal is directly limited by the system clock frequency. Since the digital counter is compared to the reference once every clock cycle, the finest adjustment of the duty cycle is equal to the length of the clock cycle. For example, 14 bit duty cycle resolution around 1 MHz operation frequency requires system clock frequency of 16.38 GHz.

The delay-line architecture utilizes the propagation delays of buffers to set the duty cycle of the output. A clock input of desired frequency is fed to this modulator, and in the beginning of each clock cycle, the modulator output is set to logic high with a latch. The rising edge of the clock then propagates through series of buffers, each of which has a known delay. The delay of a single buffer corresponds to the least

significant bit of the duty cycle command word. The delay must be equal in each buffer so that the duty cycle varies linearly in response to changes in the command word. The number of buffers through which the rising clock edge propagates to the reset port of the output latch is determined by the digital command word. With this architecture, the finest possible adjustment is limited only by the smallest propagation delay of available silicon resources, usually given by tens or hundreds of picoseconds. This is possible with a simple inverter gate, for example. The main disadvantage is that a large number of required buffers consumes large area of silicon or equivalently large amount of available FPGA resources. To implement a DPWM module with 10 bit resolution, there must be 1024 buffer elements in series. Other issue is that the propagation delay of logic gates depends on multiple variables, like silicon production process, junction temperature and used supply voltage. This is challenging if the final product is designed to be used in varying temperatures or if it is desired that the FPGA chip vendor is not fixed to just one.

3 Control Implementation

Many issues must be addressed when designing the control of a power converter. Starting from top level, the first step is the placement of the necessary devices within the physical layout of the converter. This is called control system partitioning. At the same time, a suitable way to generate the PWM control signals for the switching devices must be found, while taking into account the necessary performance as well as cost. The various measurements and their individual requirements must be considered. When the control system includes an FPGA, the required logic resource consumption of different implementations must be taken into account. This has a direct effect on the total system cost. Finally, suitable components must be chosen. In this work, the key component is the FPGA chip. Since the initial goal of the work was to find a low-cost way to implement the control, the chip selection was done in the beginning. The chosen FPGA architecture was ECP5 from Lattice Semiconductor [22], because of the price and available hardware and logic resources. The ECP5 chips include hardware digital signal processing blocks with multipliers and accumulators, and offer a relatively large number of LUTs in their price range. At the time of writing, the smallest LFE5U-12F chip with 12000 LUTs sold for less than 6$ (minimum quantity 168) in online component distributor Digikey.com.

3.1 Control System Partitioning

Digital control allows multiple different configurations in the way the control system is partitioned. The optimal partitioning offers the best balance of several factors, which include total system cost, control performance, hardware and software development effort considerations, system modularity, surge robustness and the ease of upgrading the product firmware in the field. The options for control partitioning of an AC-DC power converter have been researched [2], and their advantages and disadvantages are known. However, the research has been mainly focused on using microcontrollers and not FPGA chips. A well partitioned control system takes into consideration the special properties and also the limitations of FPGAs.

3.1.1 The Galvanic Isolation

In DC-DC power converter, the galvanic isolation between the primary and secondary sides is beneficial for the following reasons: [10, p. 142] .

1. To protect the user from electric shocks
2. To allow for more reference potentials
3. To avoid using components with unnecessarily high ratings

The AC mains voltage as well as the intermediate DC voltage in the primary side are dangerous to the user. This is why those should be isolated from the secondary side and the power supply chassis. The galvanic isolation in the LLC converter is in many ways the focus of the control partitioning. The power supply designer has

the freedom of choosing how to place the controllers and peripherals in relation to the isolation. The isolation forms a barrier for the critical control and measurement signals within the feedback loop, and this barrier must be crossed in some way. If the signal crossing the isolation barrier is a one-channel digital control signal, it is easy to use a low-cost optocoupler for the isolation. Another simple case is AC current: a simple transformer provides the means to cross the isolation. However, the output voltage is within the LLC control loop, and for that reason it should be measured very accurately and quickly. This places limitations on the ways that the output voltage could be measured across the isolation barrier. The way the control system is partitioned greatly affects the measurements and controls.

3.1.2 Controller Requirements

The LLC controller is required to have higher performance than the PFC controller [2]. This can be understood in terms of the output voltage. The output voltage has the strictest accuracy and dynamic requirements and it is also within the control loop of the LLC controller. The synchronous rectifier control adds complexity to the LLC control algorithm. Using the same estimations for required microcontroller performance as in [2], 1.2 MHz control loop frequency would require a microcontroller capable of executing at least 120 million instructions per second (MIPS). With an FPGA, there is no such a performance measurement as MIPS, but it can give some reference to the necessary FPGA system clock frequency and area. If the control algorithm is not parallelized at all, then the clock frequency must be at least 120 MHz for a 120 MIPS requirement. Since the control loop often includes instructions like saving some measurement or reference values or calculating the PID control terms sequentially, its length can usually be reduced considerably. This means computing in parallel, which possibly increases the utilized FPGA area but relaxes the clock frequency requirements. With ECP5 chips, system clock frequencies up to 400 MHz are possible [22, p. 65] while frequencies up to about 100 MHz are possible without extensive timing closure efforts.

3.1.3 Partitioning Options

There are number of ways the partitioning can be arranged with using just one or more FPGA chips, an FPGA and a DSP, or an FPGA and an MCU. In all cases, the LLC control is handled by the FPGA. DSP in this context means a microcontroller with digital signal processing capabilities that can be used for the PFC control (> 40 MIPS [2]), communication and system supervision. It is also assumed that it includes multiple channel AD-converter. An MCU is a low performance microcontroller that is only used for communication and system level supervision and optimization tasks. The possible partitioning options are:

1. FPGA in primary
2. FPGA in secondary
3. FPGA in both sides

4. FPGA in primary, DSP in primary
5. FPGA in primary, DSP in secondary
6. FPGA in secondary, DSP in primary
7. FPGA in secondary, DSP in secondary
8. FPGA in secondary, MCU in primary
9. FPGA in secondary, MCU in secondary
10. FPGA in primary, MCU in primary
11. FPGA in primary, MCU in secondary

The first two options mean that a single FPGA chip would generate the control law for both LLC and PFC converters, and also perform additional computing like system level optimizations and external communication. The third option would have an FPGA in both primary and secondary, with each controlling one of the converters. One of the FPGAs would handle system optimizations and external communication. In options 4. - 7., the LLC control law is generated by the FPGA and the PFC control law by the DSP, with different placements in relation to the isolation. In options 8. - 11., The FPGA is used to generate the control laws while the communication and system optimization is handled by the MCU.

The output voltage measurement requires an external ADC in all cases. This is because of the required accuracy and speed. Even though the low-cost DSPs usually have fairly fast and accurate integrated ADCs, required speed (2.8 MSPS) and accuracy (higher than 12 bits resolution) are not available within the cheapest DSPs. Routing the measurement samples to the FPGA at 1.2 MHz rate would create overhead in the DSP firmware point of view, and this would take away a lot of processing time from the DSP. There is no considerable advantage in placing the FPGA in the primary side since the DC-DC primary current can be easily measured over the isolation with a current transformer. These reasons strongly favor placing the FPGA in the secondary side.

Using only one large FPGA chip for all tasks is attractive in system integration point of view. Runtime optimization would be easy, since all the related variables would be within one controller. Upgrading the FPGA configuration in the field would be easy, since there would be no need to keep track of matching software and FPGA configuration versions as in options 4. - 11. Nonetheless, the system supervision, external communication and power-on sequence functionality would have to be implemented as a state-machine, requiring the use of a softcore processor like Latticemico32. Latticemico32 is an IP module, which implements a processor within the logic fabric of the FPGA [23]. It allows developing software in C programming language and running that software within the FPGA. However, to implement both control loops and a softcore processor with enough program memory and RAM would require much more FPGA resources than implementing just the LLC control. All

the measurements would require external ADCs, which together with a larger FPGA chip would raise the system cost too high. Because of the high volumes of DSPs, it is more expensive to use an external ADC than a DSP with an integrated ADC with similar performance. At the time of writing this work, the cheapest 1MSPS, 12 bit resolution ADCs costed approximately twice as much as the cheapest microcontroller with an integrated ADC of equal performance.

The advantages of using two FPGAs over using just one FPGA is the increased hardware modularity as well as reduction of necessary isolation crossings. The other FPGA could be replaced without changing the first one. The required FPGA resources are roughly the same for the control law generation. The communication between the chips will require some extra development effort as well as FPGA resources. Instead of isolating the ADC communications, isolation is needed for the internal communication bus. Nonetheless, this partitioning would be expensive because of the required ADCs and FPGA resources.

Since using an MCU for system optimization also requires using many external ADCs, the most attractive partitioning options from cost point of view are 6. and 7., which have an FPGA for LLC control law generation and a DSP for PFC control as well as for most of the measurements. A comparison of all the partitioning options is shown in Tables 1, 2, and 3.

Table 1: Partitioning options 1. - 4.

Layout	1. FPGA in primary	2. FPGA in secondary	3. FPGA in both sides	4. FPGA in primary, DSP in primary
Functions	FPGA handles controls, extra functionality	FPGA handles controls, extra functionality	Secondary FPGA controls LLC and extra functionality, primary FPGA controls PFC	FPGA controls LLC, DSP controls PFC and extra functionality
External ADCs	All measurements	All measurements	All measurements	Output voltage, output current, primary current
Switch isolation	Synchronous rectifier	LLC primary, PFC	LLC primary	Synchronous rectifier
Isolated voltage meas.	Output	Intermediate, PFC bridge	-	Output
Isolated current meas.	Output	LLC primary, PFC	LLC primary	Output
Int. comm. isolation	No	No	Yes	No
Ext. comm. isolation	Yes	No	No	Yes
Surge robustness	Least	Medium	Most	Least
FPGA resources	Most	Most	Most	Least
Field programming	Easiest	Easiest	Hardest	Hardest
Runtime optimization	Easiest	Easiest	Hardest	Hardest
Cost	Highest	Highest	Highest	Lowest

Table 2: Partitioning options 5. - 8.

Layout	5. FPGA in primary, DSP in secondary	6. FPGA in secondary, DSP in primary	7. FPGA in secondary, DSP in secondary	8. FPGA in secondary, MCU in primary
Functions	FPGA controls LLC, DSP controls PFC and extra functionality	FPGA controls LLC, DSP controls PFC and extra functionality	FPGA controls LLC, DSP controls PFC and extra functionality	FPGA handles controls, MCU extra functionality
External ADCs	Output voltage, primary current	Output voltage, output current, primary current	Output voltage, primary current	All measurements
Switch isolation	Synchronous rectifier, PFC	LLC primary	LLC primary, PFC	LLC primary, PFC
Isolated voltage meas.	Output, intermediate, PFC bridge	-	Intermediate, PFC bridge	Intermediate, PFC bridge
Isolated current meas.	PFC	LLC primary	LLC primary, PFC	LLC primary, PFC
Int. comm. isolation	Yes	Yes	No	Yes
Ext. comm. isolation	No	No	No	No
Surge robustness	Least	Most	Medium	Medium
FPGA resources	Least	Least	Least	Medium
Field programming	Hardest	Hardest	Hardest	Hardest
Runtime optimization	Hardest	Hardest	Hardest	Medium
Cost	Medium	Lowest	Medium	Medium

Table 3: Partitioning options 9. - 11.

Layout	9. FPGA in secondary, MCU in secondary	10. FPGA in primary, MCU in primary	11. FPGA in primary, MCU in secondary
Functions	FPGA handles controls, MCU extra functionality	FPGA handles controls, MCU extra functionality	FPGA handles controls, MCU extra functionality
External ADCs	All measurements	All measurements	All measurements
Switch isolation	LLC primary, PFC	Synchronous rectifier	Synchronous rectifier
Isolated voltage meas.	Intermediate, PFC bridge	Output	Output
Isolated current meas.	LLC primary, PFC	Output	Output
Int. comm. isolation	No	No	Yes
Ext. comm. isolation	No	Yes	No
Surge robustness	Medium	Least	Least
FPGA resources	Medium	Medium	Medium
Field programming	Hardest	Hardest	Hardest
Runtime optimization	Medium	Medium	Medium
Cost	Medium	Medium	Medium

The block diagrams for partitioning options 6. and 7. are shown in Figures 13 and 14. In option 6 the FPGA is placed in the secondary and the DSP in the primary side. Since both controllers generate one of the control laws, only variables related to system optimization and start-up sequence would have to be transferred over the internal communication. One such variable is the output current measurement. Assuming the measurement resolution to be 12 bits, and the system optimization task running in the DSP at an execution rate of 100 us, data transfer rate of 120 kilobytes per second is required. Even adding to this the internal communication messages and taking into account the frame size of the communication protocol, a standard 10 MHz SPI is well enough. Along with the isolated SPI communication, some isolated I/Os would be needed for communicating status information. The external communication could be referenced to the power supply chassis and sent through the FPGA over the isolated SPI, ultimately handled by the DSP.

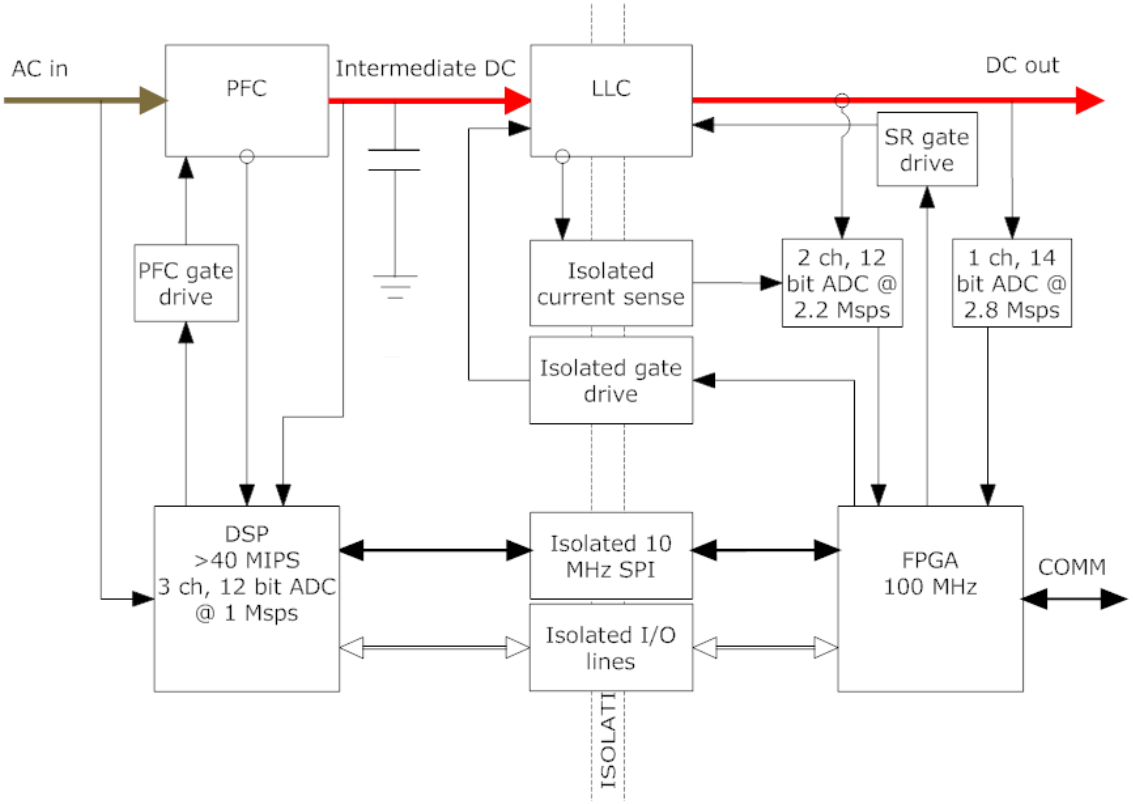

Figure 13: Block diagram of partitioning option 6.

Placing both the DSP and the FPGA on the secondary side (option 7., Figure 14) allows using unisolated digital I/Os for the internal communication, as well as some individual I/Os for communicating status information. The external communication could be referenced to the power supply chassis and handled by the DSP. This way we avoid having to route the external communication through the FPGA and again through the internal communication bus. The output current could be directly measured by the DSP, allowing the use of a less expensive external ADC for only the primary DC-DC current measurement. However, all the PFC control related measurements would have to be measured across the isolation. It should be noted that these measurements would require analog isolation. For this reason this partitioning is more vulnerable to surge voltages and currents.

In summary, partitioning option 6. seems to offer the best balance of several factors. It can be implemented with least cost and offers best protection against surge voltages and currents. If it is desired to use even higher control loop frequencies in the future, it is enough to use faster ADCs in the secondary side and a more powerful DSP in the primary side. Since the only analog measurement over the isolation is the primary DC-DC current, the measurements themselves would not need any improvements at higher control frequencies. The external communication could be taken straight from the primary side DSP through digital isolation, avoiding

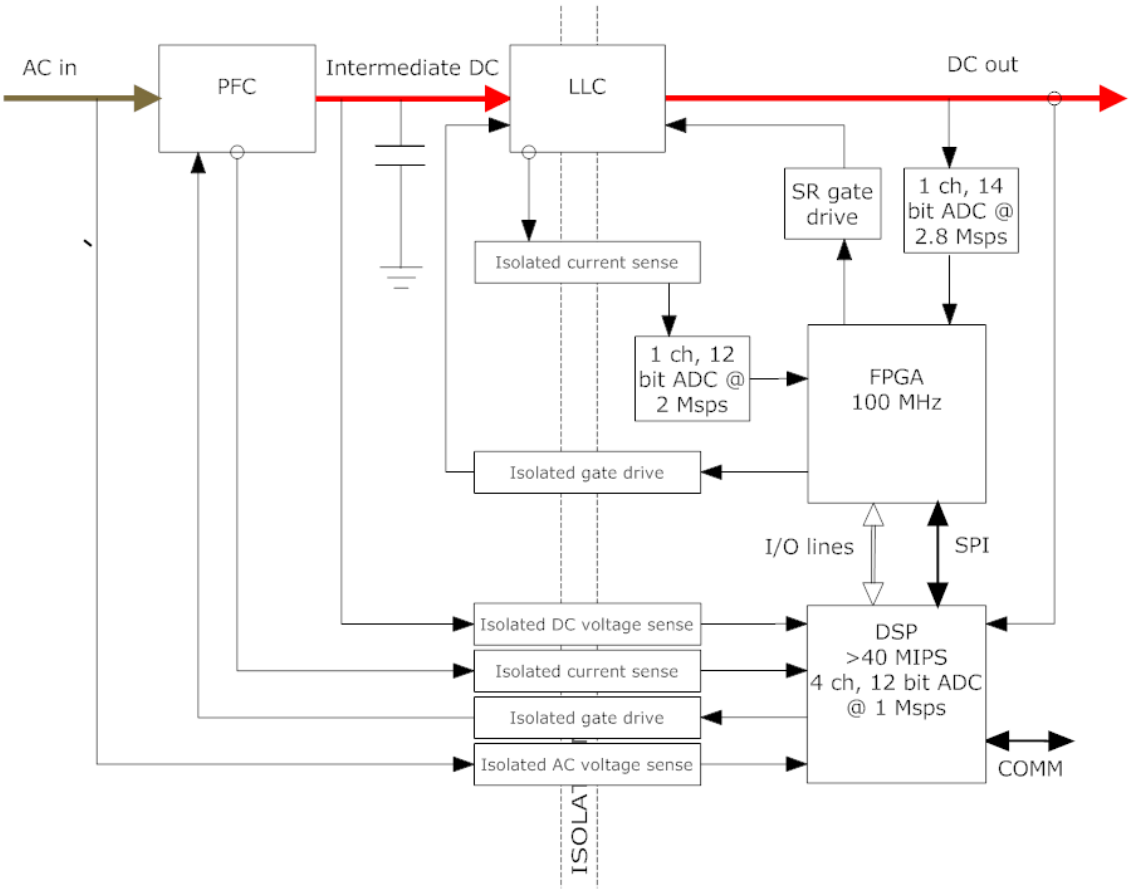

Figure 14: Block diagram of partitioning option 7.

having to route it through the FPGA, relaxing the speed requirement for the internal communication bus. However, maintaining the internal communication interface and ensuring that the controllers have matching firmware and logic configurations requires discipline and effort.

3.2 Finite State Machine Structure

Initially the LLC control law generation itself was implemented as a single logic path with no pipeline stages. This means that all the actions of the control law generation were performed between two consecutive rising edges of the system clock. This implementation minimizes the control loop delay from AD conversion to the next control command word. The maximum control rate could be found out from the place-and-route report as the maximum frequency of the system clock. The maximum system clock frequency for the initial implementation was reported to be approximately 6 MHz. However, this implementation consumed more logic resources than is available in the selected FPGA chip.

Many low-cost FPGAs including ECP5 FPGAs have integrated hardware digital

signal processing blocks, which include multipliers and accumulators [22, p. 23]. Using the hardware DSP blocks available, the amount of consumed FPGA resources decreased, but was still almost all of the resources available. To account for future additions like communication capabilites, some resources should be left available. Furthermore, using the DSP blocks without properly pipelining the data flow means that during each multiplication the operands must be routed from the logic fabric to the DSP block, and similarly back from DSP block to logic fabric. All of this happens within the same clock cycle. When there are several multiplication operations required, the routing delays add up, reducing the maximum system clock frequency considerably.

To account for these issues, the control loop was modified to a state machine form. The state machine ensured that only the operations within one state would be executed simultaneously, resulting in reduced FPGA resource consumption through resource sharing. Another advantage is the possibility to pipeline the design to allow for much higher system clock frequencies with the price of increasing the delay from AD conversion to next control command word. The pipeline registers were used within the DSP block to maximize the system clock frequency. Combined with basic operations like resizing or limiting the operands, the maximum system clock frequency for any of the states was approximately 110 MHz. The remainder of the control algorithm was split into evenly sized steps to not restrict the maximum system clock frequency. The whole control law generation could be fit into 47 steps, corresponding to 470 ns delay with a 100 MHz system clock. The PI control loops including filtering, dynamic coefficient calculation and saturation limits required 18 steps, while the secondary side synchronization algorithm required 29 steps. The simplified control algorithm flowchart is shown in Figure 15.

With control delay of 470 ns, 1.2 MHz one cycle control is possible. If it is not necessary to have one cycle control and the latency is not an issue, it is possible to convert the state machine to a pipeline and feed measurements to the controller at a maximum rate of 100 MHz. Using hardware DSP blocks reduces logic resources and allows for greater data throughput, while requiring a pipelined design. State machine design allows for debugging: the integrator register values within the PI loops and the dynamic coefficients could be sent for communication between control computation runs, unlike in combinational logic implementation. A state machine implementation allows convenient state-based branching in case of exception conditions, for example overcurrent.

3.3 Pulse Width Modulation Architecture

The two basic PWM architectures were discussed in section 2.5.3. In this work, the counter-based architecture has been chosen over the delay line based one because of its more stable operation over varying conditions, easier portability and code reuse. Furthermore, the chosen architecture is based on the synchronous architecture, with fine resolution bits achieved by clock phase-shifting using a phase-locked loop. Using synchronous logic eases static analysis and reduces PWM glitching while making it easy to port the design to a different FPGA architecture in the future

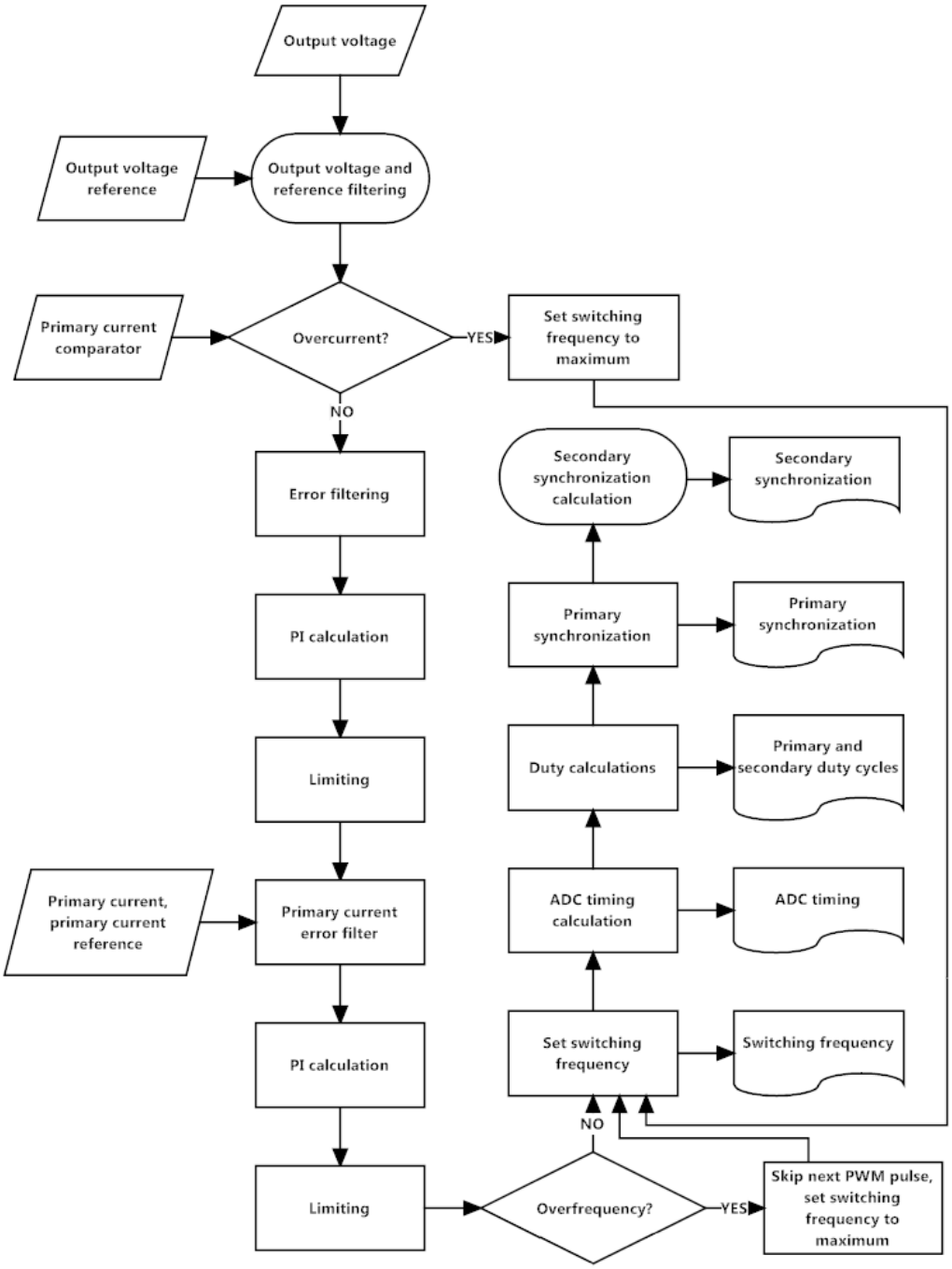

Figure 15: LLC control algorithm

[17]. The counter-based PWM architectures rely on the stability of the system clock, with low jitter and skew. The delay line based architectures rely on fixed propagation delay of available logic gates. Many power supplies must be able to function within industrial standard temperature range of -40°C to +85°C. In such a great temperature range, the gate delays in FPGAs might vary substantially, causing

the PWM resolution and linearity to degrade. The gate delays are always specific to certain FPGA architecture, which makes it challenging to port the design on another FPGA. However, almost all FPGA architectures include a phase-locked loop or a similar clock management IP, which allows creating clock signals with a fixed phase difference. The counter-based synchronous PWM architectures are readily portable to another FPGA architecture by making only the necessary configuration changes to required I/O ports and switching the PLL IP to the one provided by the chosen FPGA vendor to make the design work with the chosen FPGA.

The implemented VHDL code for the whole PWM architecture is included in appendix A. The PWM architecture is shown in Figure 16.

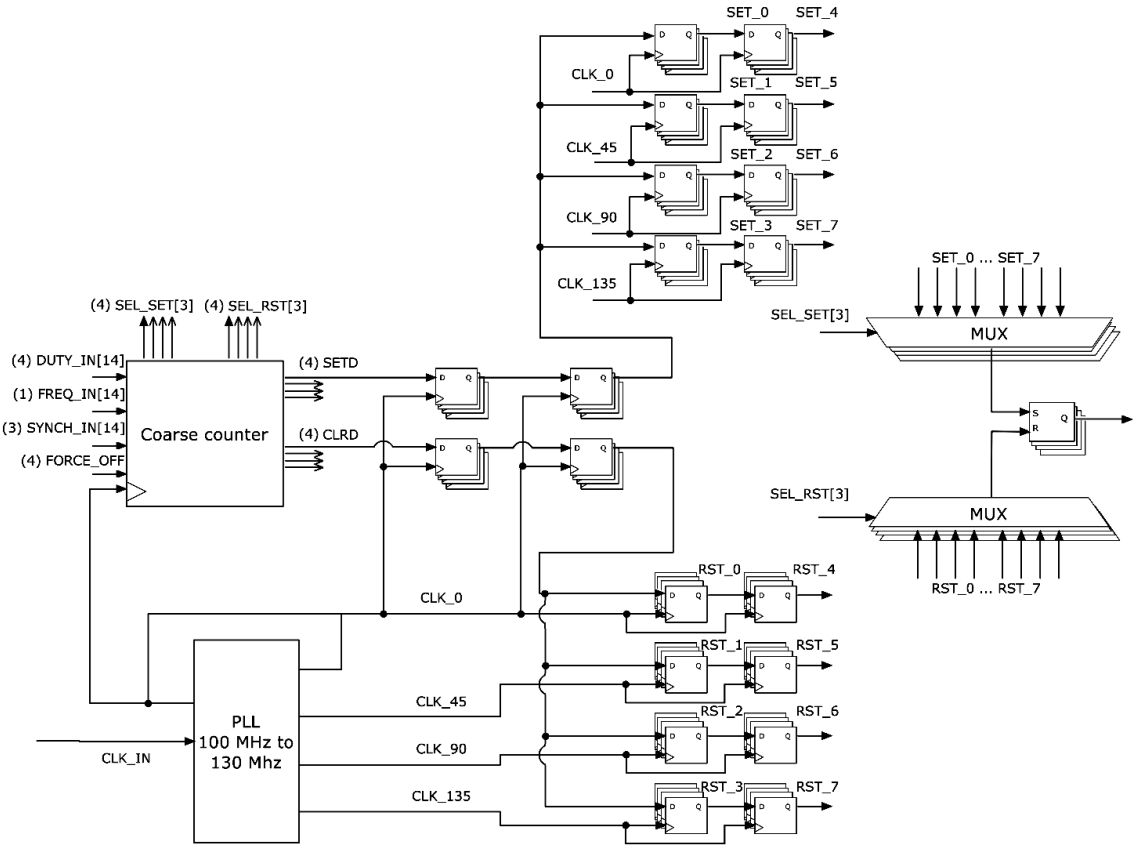

Figure 16: Synchronous shifted-clock PWM architecture

3.3.1 The Phase-shifting

The essential element of the chosen PWM architecture is the phase-locked loop block. In the ECP5 architecture, one PLL block can have up to four outputs [22, p. 18]. The input clock (CLK_IN) frequency is the same as the one used by the control law generator, 100 MHz. The PLL is configured to multiply its input clock frequency to generate four 130 MHz clocks, with 0°, 45°, 90° and 135° phase-shifts related to the 0° output clock (CLK_0, CLK_45, CLK_90 and CLK_135). The purpose of the

multiple phase-shifted clocks is to achieve higher effective clock frequency and thus higher PWM resolution than would be possible with a single clock. With low-cost FPGAs, it is very challenging to implement the necessary counter module within the PWM with clock frequency higher than 300 MHz. However, with four 130 MHz clocks, using both rising and falling edges, 8 different clock edges can be used.

$$f_{eff} = 8 \cdot f_{clk} = 8 \cdot 130MHz = 1.04GHz \tag{32}$$

$$\rightarrow t_{eff} = \frac{1}{f_{eff}} \approx 0.96ns. \tag{33}$$

An effective clock frequency of 1.04GHz is obtained. The equivalent time resolution of the design is then 0.96 ns, or around 10 bits at the control frequency of 1 MHz. The clock frequency of the phase-shifted clocks was chosen based on the design requirement of 1 ns time resolution. With LFE5UM-45F FPGA chip, after place and route phase, CLK_0 was reported to be limited to the maximum frequency of 154.083 MHz. This frequency limit is caused by the coarse counter module: the sequential logic within that module cannot be clocked over 154.083 MHz. Based on this report, the maximum time resolution for this PWM architecture on this FPGA chip would be approximately 0.8ns.

With flip-flops that are configured to be sensitive to either rising or falling edge of one of the clocks, the SETD signal and the CLRD signal can be selectively propagated towards the output. The first two flip-flops in the pathway are sensitive to the rising edge of CLK_0. These flip-flops ensure that there is no uncertainty related to the instant when the SETD or CLRD signal crosses the clock domain between CLK_0 and the rest of the clocks. If SETD or CLRD is set to logic high at the first rising edge of CLK_0, at the third rising edge of CLK_0 the signal propagates to the inputs of the last flip-flops before the multiplexer. For example, in the case of low to high PWM output transition illustrated in Figure 17, the output is set to logic high at the falling edge of CLK_0. in Figure 16, this corresponds to SEL_SET[3] having binary value ”100”, which selects SET_0 to the output. With SEL_SET[3] binary value ”111”, the rising edge of CLK_45 would be chosen corresponding to a single LSB delay in relation to the third rising edge of CLK_0. A value of ”000” would be the fourth rising edge of CLK_0, which is full 8 LSB delay steps from the third rising edge. With this synchronous design, we obtain three fine-resolution bits within the clock cycle of CLK_0.

3.3.2 The Counter

The coarse counter counts rising edges of CLK_0. Furthermore, all the counting and all the actions of the counter are performed synchronously to CLK_0, in low resolution. High resolution is achieved by separately handling the three fine resolution bits in an appropriate way, which corresponds to advancing and delaying the SETD and CLRD signals as needed. The coarse counter increases until a maximum value, which is calculated again within each PWM cycle. Since the PWM frequency is dynamically configurable with high resolution, the counter maximum cannot stay

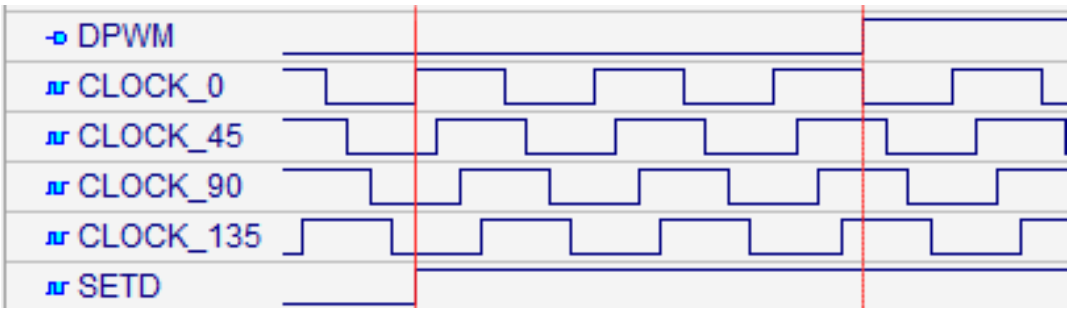

Figure 17: Screenshot from shifted clock simulation

fixed for a constant frequency setting, if the LSBs are not all zeros. For example, PWM carrier frequency of 507.8125 kHz corresponds to FREQ_IN[14] set to ”00 1000 0010 0000”. This means that the counter would increment its value until 260.00 every PWM cycle. However, if the carrier frequency was set to 499.7597 kHz, FREQ_IN[14] would be ”00 1000 0010 0001”. Now the LSBs are not all zeros, and the required counter value would be 260.125. This can be solved by delaying the individual PWM pulses 0.125 clock cycles every PWM cycle, and count to 261 every eight PWM cycle, otherwise to 260. At the 8th PWM cycle, the PWM pulse is advanced 0.875 clock cycles and the process is repeated.

Delaying and advancing the individual PWM pulses requires that the set and reset signals are determined with high resolution. These signals are also calculated again within each PWM cycle. The operation principle of the implemented counter module is shown in Figure 18.

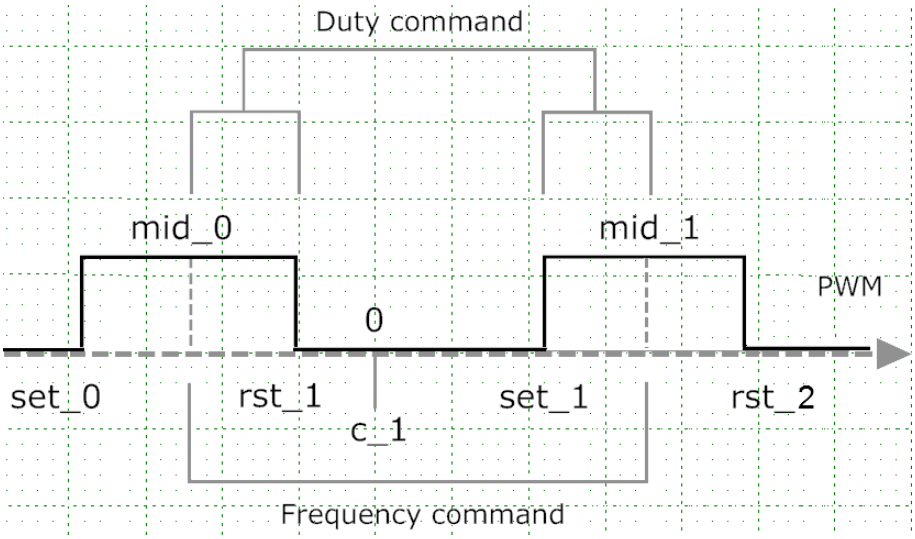

Figure 18: High-resolution PWM operation

Middle points of the PWM pulses (mid_0, mid_1) are calculated in high resolution. The next middle point depends on the previous middle point, counter maximum as

well as the current carrier frequency (period) command word FREQ_IN[14]. The new period is added to the last middle point, and the current counter maximum (c_1) is subtracted to give the new middle point (mid_1). This calculation is performed at the point when the counter value equals the current middle point with the fine resolution bits truncated away, equivalent to the previous rising edge of CLK_0. The distance between the middle points corresponds to the frequency command word. It must be noted that the calculated middle points are not necessarily the actual middle points of the PWM pulses. If the duty command does not stay unchanged, the latter PWM pulse half can be longer or shorter than the first. The coarse counter maximum (c_1), also the zero of the next counter run, is also determined at the previous middle point (mid_0). The counter maximum is obtained by dividing the difference between the rising edges of CLK_0 previous to consecutive high-resolution middle points by two and truncating the result.

From the calculated high resolution middle point, the next reset instant (rst_1) is calculated by adding half of the current duty cycle command word DUTY_IN[14]. The next set instant (set_1) must then be the next high resolution middle point (mid_1) minus half of DUTY_IN[14], yielding the desired pulse width as well as the carrier period. The implemented PWM is then an up-count PWM, with the PWM pulses around the middle point of the carrier. The counter module updates the values of set and reset signals only during rising edges of CLK_0. The fine resolution bits are simultaneously updated and used to select the outputs of appropriate flip-flops within the signal paths of SETD and CLRD.

In addition to duty cycle and frequency, the synchronization of the channels relative to the master channel can be configured in high resolution. This is an important feature in bridge circuits within power supplies, where the high-side and low-side switches must be out of phase. When the master counter reaches the middle point (coarse), the counters of all other channels are assigned the value of 11 most significant bits of respective synchronization command words (SYNCH_IN[14]). This way the channel counters can be delayed or advanced relative to the middle point of the master counter once every PWM cycle. The duty cycles of each channel can also be separately configured, but the frequency is the same for all channels. The LSBs of SYNCH_IN[14] are used to advance the set signal of the respective channel 0-7 high resolution steps. In similar way as the channels are synchronized, there is an option to synchronize an AD-conversion to a certain point along the carrier of the master PWM channel. This is also a requirement in power supplies, where some current measurements must be performed at a point where the current represents an average value. Measuring the current during the switching instants can result in noisy and distorted measurements.

The individual channels can be separately disabled with FORCE_OFF signal. The SETD signal is set to logic high only in the case that the FORE_OFF is not up. This is done with combinational logic statements, which allow enabling and disabling individual PWM pulses. This feature is useful for current and power limiting. If the measured current is greater than the peak current limit, the PWM pulses can be disabled starting from the very next pulse, until the measured current has again fallen within safe limits. With similar combinational logic, it is also possible to ensure

that there can be no shoot-throughs in the high-side and low-side switches. Using exclusive-or (XOR) and AND gates, the PWM signals can be configured to never be at logic high simultaneously. This can be very useful feature in the early phases of prototyping, when the correct control functionality is verified with actual power supply hardware.

The number of channels can be easily increased or decreased. This requires reconfiguring the coarse counter module VHDL code. Adding channels only adds to the number of registers, multiplexers and the output latches. The number of PLLs and coarse counter modules doesn't increase, since all the channels use those same modules. Similarly, the number of synchronized ADC measurements can be easily increased and even added to other channels than the master channel without having to add any additional modules. However, in this implementation the ADC synchronization has been designed to use only the coarse resolution bits.

3.3.3 The Asynchronous Output

The whole PWM architecture implementation is synchronous except for the very last part. In that part, the appropriate flip-flop output is selected by the asynchronous 8-input multiplexer. In the output of the PWM module, there is an asynchronous set-reset latch. The latch toggles its output based on its input signals, set and reset. A rising edge of set input causes the latch to output a logic high. Correspondingly, a rising edge of the reset causes the latch's output to reset to logic low. The ECP5 architecture doesn't include hard set-reset latches in the logic fabric, which means they have to be implemented using combinational logic statements.

3.4 Synchronous Rectifier Control

To minimize the losses caused by the switching of the secondary side rectifier circuit, the switching must take place close to the instants of the zero crossing points of the secondary current. This way the product of the instantaneous current through the switches and the voltage over them is approximately zero. To properly synchronize the secondary switches to the primary side switches, the relationship between the phase of the secondary current and the voltage applied to the primary resonant circuit must be calculated. Since the zero crossing points of the applied voltage take place during switching instants on the primary side, the phase difference is directly related to the required synchronization between the primary side and secondary side switches.

3.4.1 Secondary Synchronization

In the following analysis, the goal is to find the phase difference between the primary voltage and the secondary current. As it is shown in Figure 6, the secondary current I_s goes through R_{AC} while the primary voltage is defined as the voltage over R_{AC}. V_{AC} is the square wave which is applied into the resonant tank by the primary bridge circuit. Since there is no phase difference between current and voltage in a pure

resistor, it is enough to find the phase difference between primary voltage V_p and V_{AC}. From Equation 20:

$$\angle V_p - \angle V_{AC} = \angle \frac{-\omega^2 C_r L_m R_{AC}}{j(\omega L_m - \omega^3 C_r L_m L_r) - \omega^2 C_r R_{AC}(L_m + Lr) + R_{AC}} \tag{34}$$

$$\rightarrow \angle V_p - \angle V_{AC} = \pi - \angle(j(\omega L_m - \omega^3 C_r L_m L_r) - \omega^2 C_r R_{AC}(L_m + Lr) + R_{AC}) \tag{35}$$

$$\rightarrow \angle V_p - \angle V_{AC} = \angle(j(\omega L_m - \omega^3 C_r L_m L_r) - \omega^2 C_r R_{AC}(L_m + Lr) + R_{AC}) \tag{36}$$

$$\rightarrow \angle V_p - \angle V_{AC} = arctan(\frac{\omega^3 C_r L_m L_r - \omega L_m}{\omega^2 C_r R_{AC}(L_m + Lr) - R_{AC}}). \tag{37}$$

As the controller operates on the switching period t_s rather than angular frequency, Equation 37 becomes:

$$\angle V_p - \angle V_{AC} = arctan(\frac{8\pi^3 C_r L_m L_r - t_s^2 2\pi L_m}{t_s 4\pi^2 C_r R_{AC}(L_m + Lr) - t_s^3 R_{AC}}). \tag{38}$$

From Equation 38 it is evident that the required complexity of the secondary synchronization control algorithm is relatively high. To calculate the PWM synchronization in high resolution, the control algorithm must perform some multiplications and subtractions. Additionally, the control algorithm must compute values of *arctan* function. To simplify the control calculation, *arctan* can be estimated with piecewise linear approximation. However, this will introduce some error to the synchronization. The amount of acceptable error in the synchronization will have to be tested with the actual power converter hardware and the implemented control. Besides the synchronization itself, the optimum duty cycle for the secondary side switches must be calculated separately. While the primary switches are always operated with a duty cycle close to 50%, the secondary duty cycle must be reduced in the case the switching frequency is below resonant frequency. As can be seen in Figure 9, the secondary switches should conduct only for a portion of the primary switching cycle.

3.4.2 Dynamic Range Requirement

To find out whether it is feasible to use integers for the arithmetic, the necessary dynamic range must be determined. As the typical values for reactive components like inductors and capacitors are of the order of 10^{-6} or less, and the maximum switching period with the minimum control frequency of 375 kHz is equivalent to about 2770 clock cycles of 0.96 ns resolution, it is inevitable that arithmetic operations on both very small and very large numbers must be performed. Multiplying or subtracting such numbers with each other cannot be avoided by rearranging the above equation. Using the case mentioned, the required dynamic range is at least the following:

$$DR = \frac{2770^3}{10^{-6}} \approx 2 \cdot 10^{16} \approx 2^{54}. \tag{39}$$

It would require then binary width of at least 54 to accurately represent the operands and the results of intermediate calculations of the secondary side synchronization. Arithmetic operations on binary words of this width would consume excessive amount of FPGA resources and place strict restrictions on the maximum control rate. For these reasons it is necessary to use floating-point arithmetic in the secondary side synchronization algorithm.

3.5 Floating-Point Arithmetic

In floating-point representation, the sign bit, exponent and mantissa have separate fixed-width bit fields reserved in the binary word. The mantissa contains the significand of the number, while the exponent is used to scale the number. The width of the mantissa defines the available resolution. Because of this, the largest integer that can be precisely represented in floating-point format is dependent on the mantissa bit width. Using floating-point representation of numbers allows for much greater dynamic range than with integers with the cost of reduced precision. However, absolute precision is not always required. In this work, the PWM duty resolution is approximately 10 bits around the control frequency of 1 MHz. The controller output doesn't need to be designed for higher resolution than that. For this reason, floating-point arithmetic is an appropriate choice to calculate the secondary synchronization efficiently. The implemented VHDL code for all the floating-point modules is included in appendix B.

3.5.1 Single-Precision Representation

Using the IEEE 754 single-precision format, the sign bit, exponent and mantissa have separate bitfields reserved in a 32 bit word [24, p. 9]. The significand of single-precision floating-point number is called the mantissa and is represented with 23 bits. It is assumed that the most significant 24th bit is always 1, and that is why it is not explicitly kept in the actual binary word. Using s, e, and m for sign bit, exponent bits and mantissa bits respectively, the decimal value represented by the floating-point binary word is as follows:

$$(-1)^s \cdot 2^{(\sum_{n=0}^{7} e_n 2^n)-127} \cdot (1 + \sum_{n=0}^{22} m_n 2^{n-23}), \tag{40}$$

where n is the index of bits, 0 being the least significant bit. Thus, the significand can contain an equivalent decimal value between 1,0 and 2,0 - 2^{-23}. The exponent is represented by 8 bits. It is defined that the exponent is offset by an integer of 127, meaning that the exponent can have a value between 2^{-127} and 2^{128}. However, exponents -127 and 128 are reserved for special values. The range that can be used is then between 2^{-126} and 2^{127}. The dynamic range is then:

$$DR \approx 2 \cdot 2^{2^8-2} = 2^{255} \approx 10^{77}. \tag{41}$$

The above range is well enough for the secondary synchronization calculations.

3.5.2 Number Format Conversions

An unsigned 32 bit binary integer corresponds to following decimal value:

$$\sum_{n=0}^{31} b_n 2^n, \tag{42}$$

where b_n are the individual bits of index n in the binary word. To convert this unsigned integer to floating-point format, we must represent it in a form similar to Equation 40. Let p be the index of the most significant non-zero bit within the unsigned integer:

$$\sum_{n=0}^{31} b_n 2^n = (-1)^0 \cdot 2^p \cdot (1 + \sum_{n=0}^{p-1} b_n 2^{n-p}). \tag{43}$$

It is easy to see then that the bit fields of the corresponding floating-point number are:

$$s = 0 \tag{44}$$

$$\sum_{n=0}^{7} e_n 2^n = p + 127 \tag{45}$$

$$\sum_{n=0}^{22} m_n 2^{n-23} \approx \sum_{n=0}^{p-1} b_n 2^{n-p}. \tag{46}$$

As Equation 45 shows, the exponent is simply the binary representation of the index of the most significant non-zero bit, with an offset of 127. Equation 46 holds precisely only in the case of the most significant non-zero bit being 23. If it is for example 27, only 22 bits starting from 26 are included in the floating-point number. This shows in practice how the floating-point format loses some of the precision. However, if the most significant bit is less than 23, not all bits of the mantissa can be filled. The remaining unassigned mantissa bits are then set to zero.

The conversion from floating-point to unsigned integer format works in a similar way, but care must be taken because not all floating-point numbers can be converted. The convertable range is the range of positive numbers between 0 and 2^{32}-1. First, the sign bit is set to zero. The corresponding decimal value is calculated from the exponent bits, subtracting the offset. At this point the result is checked to be limited to the range of 0 to 31. This gives the index of the most significant non-zero bit. If this is not bit 0, then the lower value bits are assigned the values of the most significant bits of the mantissa of the floating-point number. If the most significant non-zero bit of the integer is over 23, then the remaining lower value bits are set to zero.

3.5.3 Subtraction

Feasibility of using FPGA logic for floating-point calculations have been studied [25], and many floating-point adders and multipliers have been implemented on FPGAs [26] - [27]. In this work, multiplication and subtraction are necessary, even though subtraction is equivalent to addition with the sign of one of the operands inverted. In spite of available floating-point units, custom units are designed and implemented. This is because of the specific requirements of maximum control computation latency as well as the 100 MHz system clock frequency that was chosen for the controller.

The basic steps of the floating-point addition or subtraction are the following [28]:

1. Calculating the difference of exponents
2. Shifting right the mantissa of operand with smaller exponent
3. Adding or subtracting the mantissas depending on the sign bits
4. Normalizing the result
5. Rounding the result

The implemented steps are shown in Figure 19. The first steps are equivalent to above steps 1. and 2. After that, the necessary operation (add or subtract) is determined based on the sign bits of the operands. The difference of exponents is used to determine the larger operand, the final sign bit as well as the possibility of operands being equal, in which case a zero result must be separately handled. In case of addition, there is a possibility for overflow and that is properly taken care of. The result of the operation saturates to minimum or maximum in case of overflow. Finally, the result is normalized. For example, subtracting two numbers with equal exponents that are close to each other in value results to binary word with only couple of the lower bits being non-zero. In that case, the radix point must be shifted to the right side of the most significant non-zero bit, while decreasing the exponent of the result accordingly. The rounding step mentioned above is implemented simply as truncation, which decreases logic resource consumption and allows for greater efficiency while adding quantization noise less than 1 LSB.

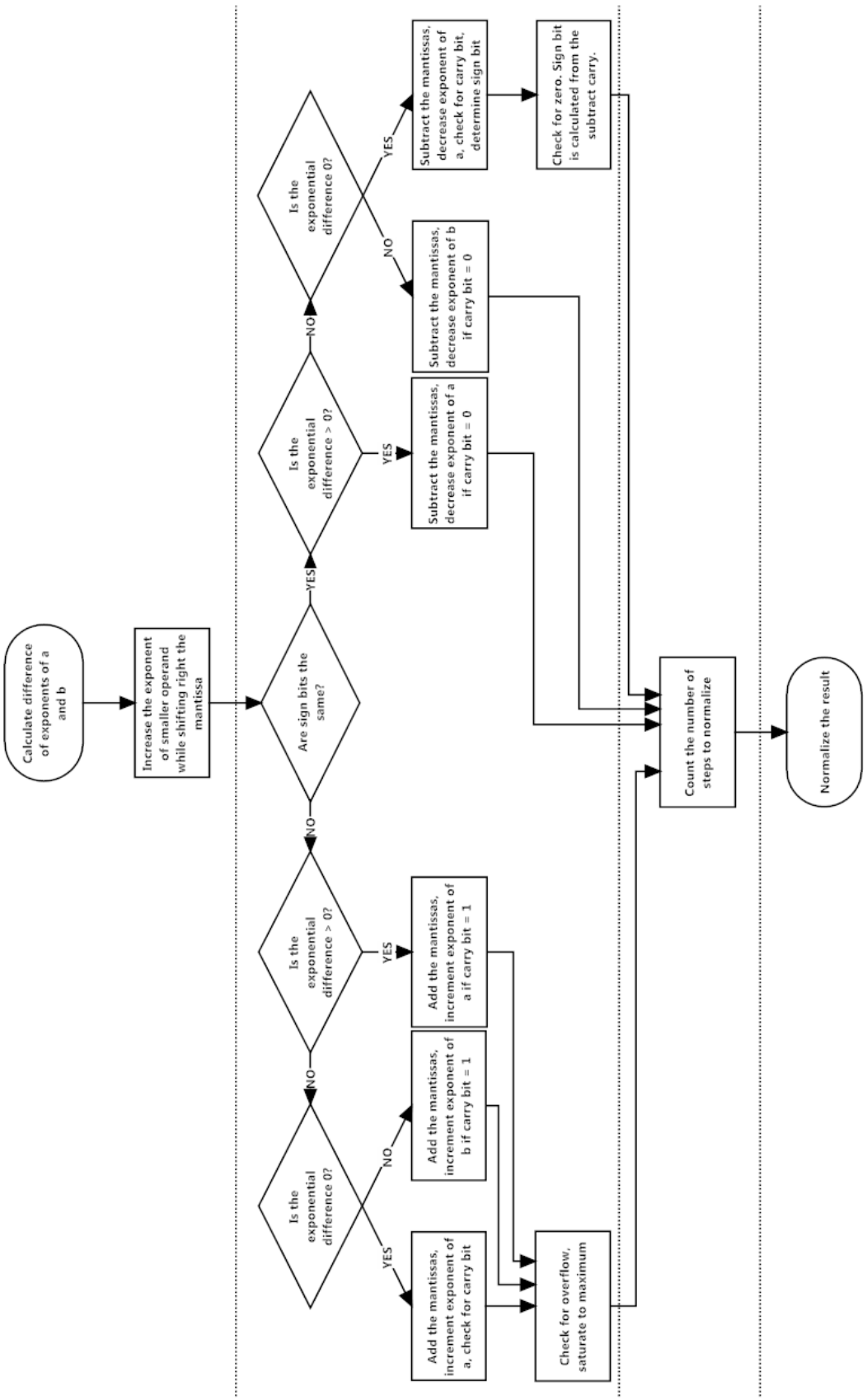

Figure 19: Flow chart of floating-point subtraction algorithm.

Floating-point subtraction algorithm is more complex than floating-point multiplication. Even though multiplication itself consumes more logic resources than subtraction, subtraction algorithm has to take into account initial normalization, finding the larger operand as well as keeping track of the possible carry bit from the mantissa subtraction or addition. Since the total control computation latency is limited, and the operations previous to any floating-point arithmetic already consumes a relationally large portion of the total budget, the subtraction algorithm must be carefully pipelined. The location of registers are shown as the dashed lines in Figure 19, and were chosen in such a way that the sequential logic between any two registers could be clocked to at least to 100 MHz. As a result, a five stage pipeline was implemented. After assigning the operands, the result is ready after five clock cycles. As can be seen in Equation 38, at least two subtractions are required for the secondary synchronization calculations. The performance and logic resource consumption is summarized in Table 4.

Table 4: Implemented floating-point subtraction module.

	24 mantissa bits	18 mantissa bits
Pipeline stages	4	4
Consumed LUTs	1028	846
Maximum clock frequency (MHz)	104.6	120.9
DSP blocks used	-	-

3.5.4 Multiplication

While floating-point subtraction requires multiple sequential steps, multiplication requires only few. The sign bit, exponent and mantissa can be calculated in parallel, while only the normalization and saturation steps must be performed sequentially. Floating-point multiplication of operands a and b is calculated with the following equations [25]:

$$s = XOR(s_a, s_b) \tag{47}$$

$$m = m_a \times m_b \tag{48}$$

$$e = e_a + e_b - 127 + (1) \tag{49}$$

The increment of one in Equation 49 is performed in case the result of the multiplication is equal to or greater than 2. As can be seen from above equations, besides the mentioned increment, there is no common terms in the equations and thus each bitfield of the result can be calculated separately in parallel. The effect of the increment is that the final result must be normalized, which must be done after

all above steps have been completed. Additionally, the result must be checked for overflows and underflows. The steps of the implemented multiplication algorithm are shown in Figure 20.

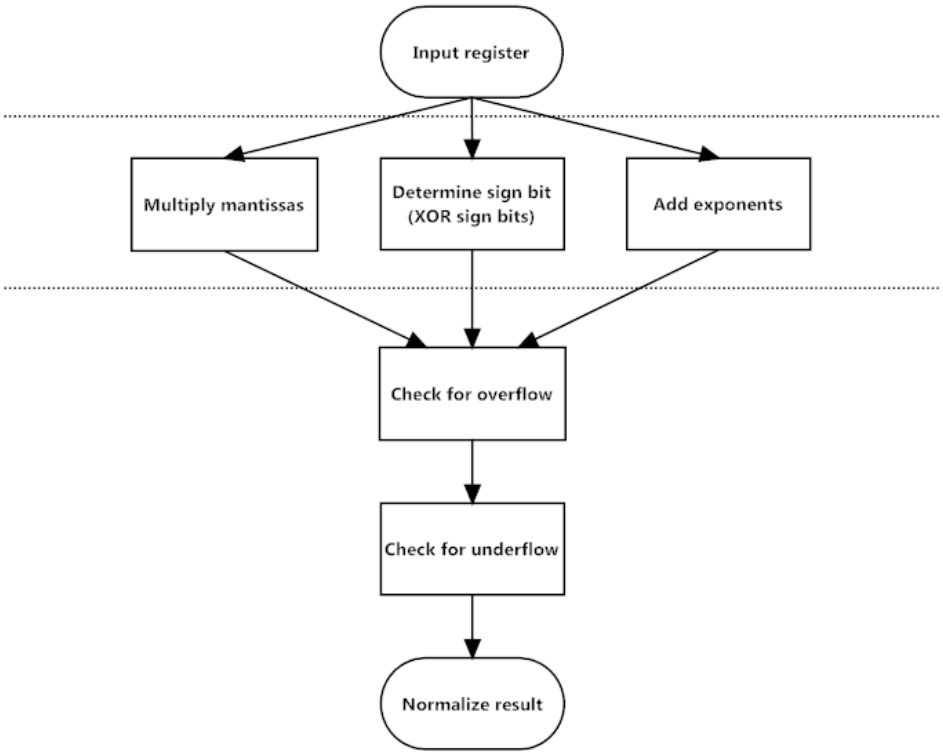

Figure 20: Flow chart of floating-point multiplication algorithm.

Table 5: Implemented floating-point multiplication module.

	24 mantissa bits	18 mantissa bits
Pipeline stages	3	3
Consumed LUTs	89	89
Maximum clock frequency (MHz)	110.5	184.0
DSP blocks used		
MULT18x18D	4	1
ALU54B	2	0

The input, pipeline and output registers were enabled to maximize the system clock frequency. The locations of the registers are marked as dashed lines in Figure 20. To further optimize the resource consumption and computation speed, the operands were truncated to 18 bit resolution. This allowed utilizing only one of the 18x18 hardware multipliers (MULT18x18D) available within one sysDSP slice of the ECP5

architecture [22, p. 18]. Again, the final rounding is accomplished with a simple truncation. The resource consumption and performance summary of 24 bit and 18 bit resolution multiplier modules is shown in Table 5.

3.5.5 Estimation of Multiplicative Inverse

As is evident from Equation 38, floating-point division is also necessary to compute the required angle. However, division is a complex and resource-hungry operation. It is much more efficient in terms of consumed FPGA resources as well as computation speed to find the multiplicative inverse of the divisor and then multiply the operands. Actually calculating the multiplicative inverse would also take many clock cycles, so an estimation was chosen in this work. 16 precalculated points were saved in an array, which was indexed with the 4 most significant bits of the operand. With the next 7 lower bits, a linear approximation was used between the precalculated points to estimate the multiplicative inverse function. With this approach, the achieved resolution for the operation was approximately 9 bits. The precalculated points as well as the linear coefficients were chosen in a way to balance the approximation error over the range of [1, 2[. The VHDL model consisting of the implemented estimation function and a testbench feeding input data was implemented, and the results were analyzed with MATLAB. The relative approximation error is shown in Figure 21.

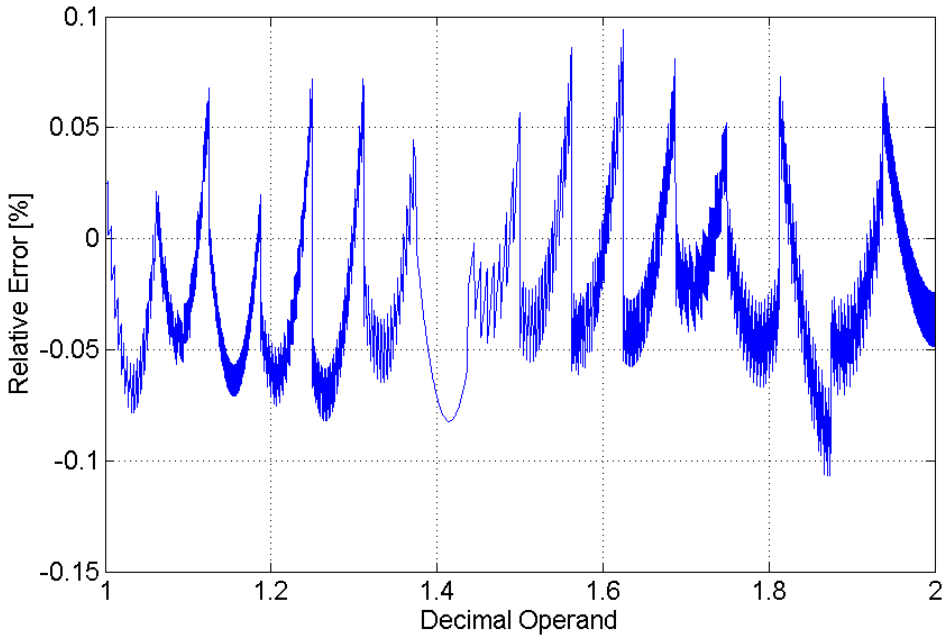

Figure 21: Approximation error of multiplicative inverse.

As the relative error is about 0.1%, this is the part of the floating-point arithmetic that introduces most of the error. However, since the secondary synchronization command is always in the range of 0 to 1000 nanosecond steps, the maximum error

because of this estimation would be only one step. The effect of this would have to be verified with the actual power converter hardware and implemented control. The multiplicative inverse estimation was designed to fit within a single clock cycle of the state machine clock. One of the available 9x9 hardware multipliers (MULT9X9D) [22, p. 18] were used. The consumed resources are shown in Table 6:

Table 6: Implemented estimation of multiplicative inverse.

	11 mantissa bits
Pipeline stages	1
Consumed LUTs	18
Maximum clock frequency (MHz)	115.0
DSP blocks used	
MULT9X9D	1

4 Simulations

A complete behavioral VHDL simulation model was constructed to test and verify the functionality of the controller and the submodules. The simulated cases include static regulation as well as dynamic response to step changes in load and output voltage reference setting. The switching waveforms and the timings are presented. The operation of individual modules such as the PWM module and the floating-point arithmetic modules is also presented in the section. For the LLC converter, VHDL model with the following parameters was used:

Table 7: LLC converter parameters.

Resonant inductance	2.9 μH
Resonant capacitance	35.3 nF
Magnetizing inductance	18.6 μH
Transformer turns ratio	3.75
Resonant frequency	500 kHz
Minimum frequency	375 kHz
Maximum frequency	1.2 MHz
Output capacitance	1 mF
Input voltage	400 V
Output voltage	48 V

4.1 Start-Up And Static Regulation

In this simulation, the power converter model was run with zero initial voltages and currents. The output voltage reference was set to 51 V and the output load resistance to 1.5 Ω. The results of the VHDL simulation were imported to MATLAB for plotting.

in Figure 22, it is shown that the output voltage rapidly reaches the reference level with small overshoot. As the load is close to the nominal, the switching frequency settles close to the resonant frequency of 500 kHz. The initial switching frequency is selected to be 750 kHz, from where it decreases towards lower frequencies and equivalently higher resonant tank gain to raise the output voltage to the desired level. Even though the resonant inductor current envelope in this case seems to stay within reasonable limits, it is possible for very large current peaks to appear in the beginning of a cold start. This is why it is necessary to have a special start-up routine implemented within the control.

Figure 23 shows the output voltage ripple in static conditions. In this simulation, the peak-to-peak output voltage ripple in static conditions was less than 20 mV. However, the PFC stage introduces some ripple to the intermediate DC bus voltage, which was not taken into account. The actual output voltage ripple with the power converter hardware could be somewhat higher for that reason.

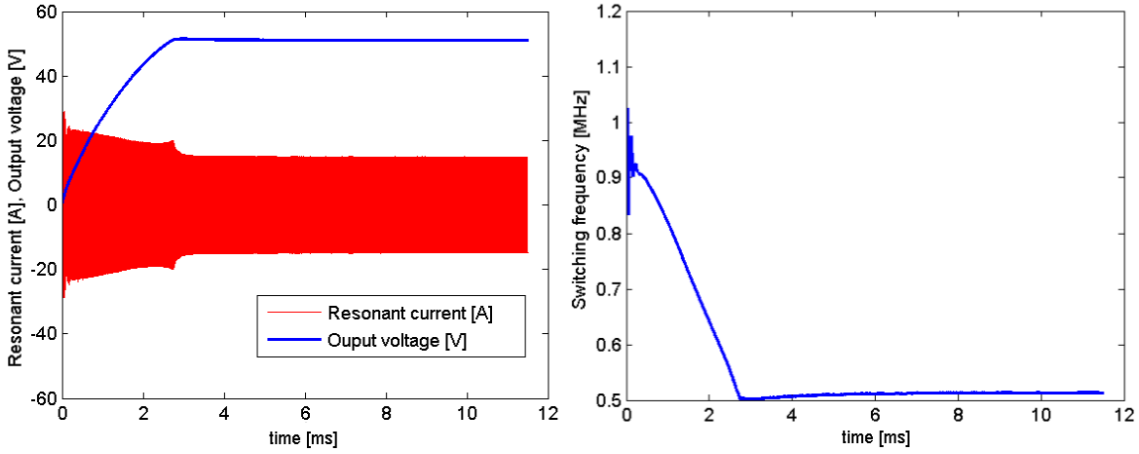

Figure 22: Output voltage, resonant inductor current and switching frequency in start-up

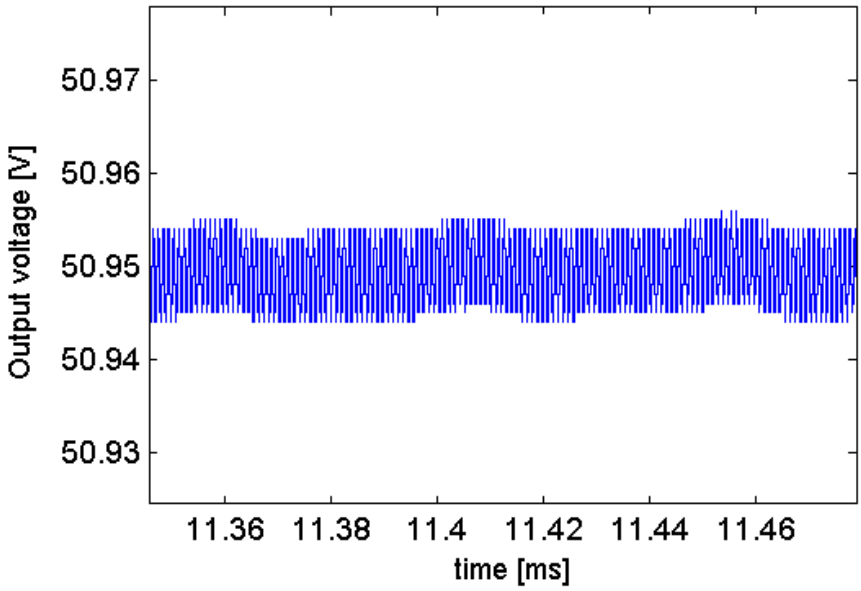

Figure 23: Output voltage ripple in static operation

4.2 Dynamic Response

In this simulation, the power converter dynamic response was studied by introducing step changes first to the load resistance and then to the reference voltage. In the first test, the load resistance was increased from 1.5 Ω to 3 Ω. This corresponds to step reduction of load by 50%. After the output voltage had stabilized, the load resistance was reverted back to 1.5 Ω, corresponding to an increase of the output power by 100%.

As can be seen in Figure 24, the output voltage peaks immediately after the

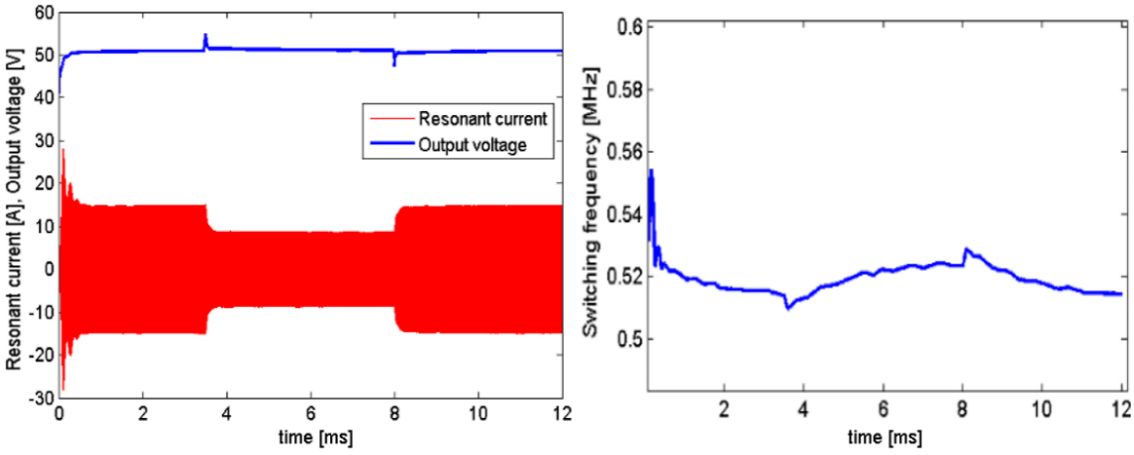

Figure 24: Step changes in load

load steps before settling back towards the reference level. In the first load step, the output voltage reaches 54.8 V. In the second load step, the output voltage drops to 47.4 V. The load level and the performance of the controller can be observed from the envelope of the resonant inductor current. Immediately after the load step changes, the resonant inductor current amplitude settles to the required level. As the controller is operating close to the resonant frequency, where the gain curves are relatively steep, the required change in the switching frequency is quite limited.

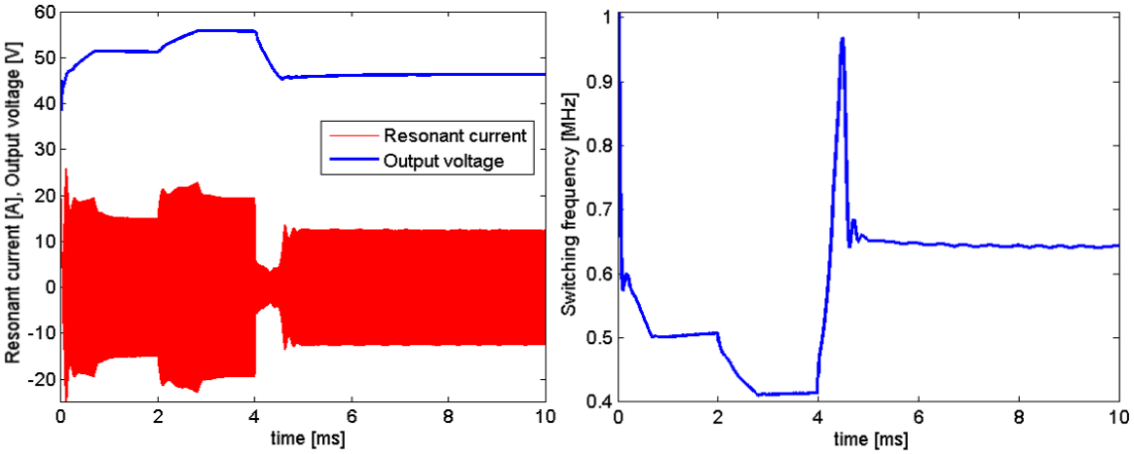

Figure 25: Step changes in reference voltage

In the second test, step changes were introduced to the reference voltage. The initial reference was the same as in previous simulations, 51 V. The reference was then raised to 55.6 V, after which the reference was finally set to 46.4 V. Figure 25 shows the power converter responses. In the first two steps, the output voltage and the controller output settle quickly. However, the final step causes a spike to the controller output, which then settles towards the required level. Some ripple can

be seen in the switching frequency, which also appears in the output voltage. This could be due to unoptimally tuned PI loops within the controller.

4.3 Pulse Skipping

In this simulation, the protective mode of the controller is demonstrated. In case of overcurrent or too high switching frequency from the controller, the switch drive pulses can be disabled for a period of time. In this case, the reference voltage was set to a low value, implying a high frequency output from the controller. As the controller reaches the maximum frequency, the protection is activated and the switching is disabled to disrupt the transfer of power in the controller and let the capacitances discharge. in Figure 26, this appears as consecutive periods of activity and inactivity in the primary side switch. The resonant inductor current appears as bursts of current, causing the output voltage to stay approximately at the reference level but introducing a relatively large ripple amplitude of 400 mV peak-to-peak.

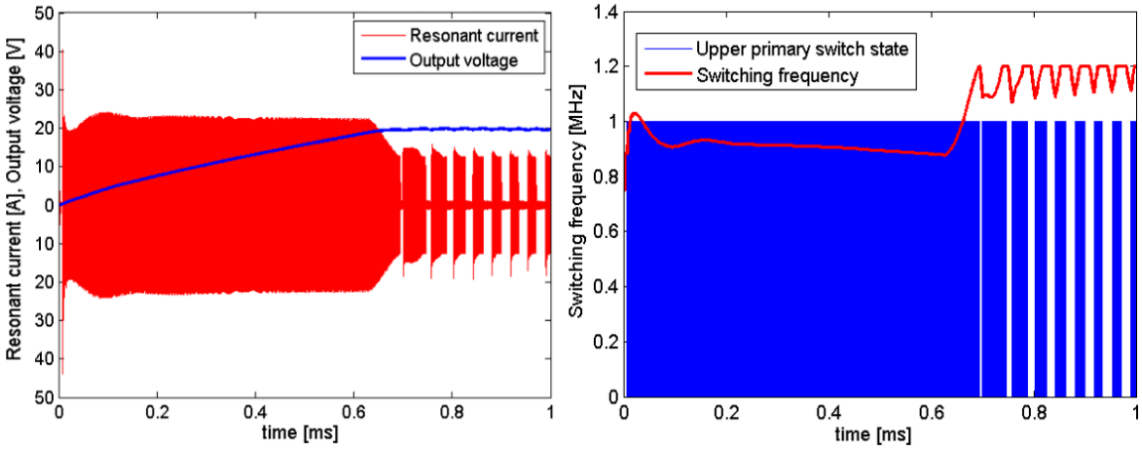

Figure 26: Burst mode

4.4 Synchronous Rectifier Control

Figure 27 shows the primary and the secondary gate pulses on same time line. The scaled-down secondary current is also presented on top of the secondary gate pulses. As was mentioned in section 2.3.1, the secondary switching should be timed to the instants when the secondary current is zero to avoid switching losses. Additionally, the switches should only conduct current to one direction to function similarly to a rectifier bridge. To ensure this, the switch pulse durations should never exceed the resonant period of the converter. In Figure 27a., the converter is operated at the resonant frequency. The secondary side pulses are approximately in phase with the primary side pulses. The pulse widths of the secondary switches are only slightly shorter than the primary side switches, and ZCS conditions are approximately achieved.

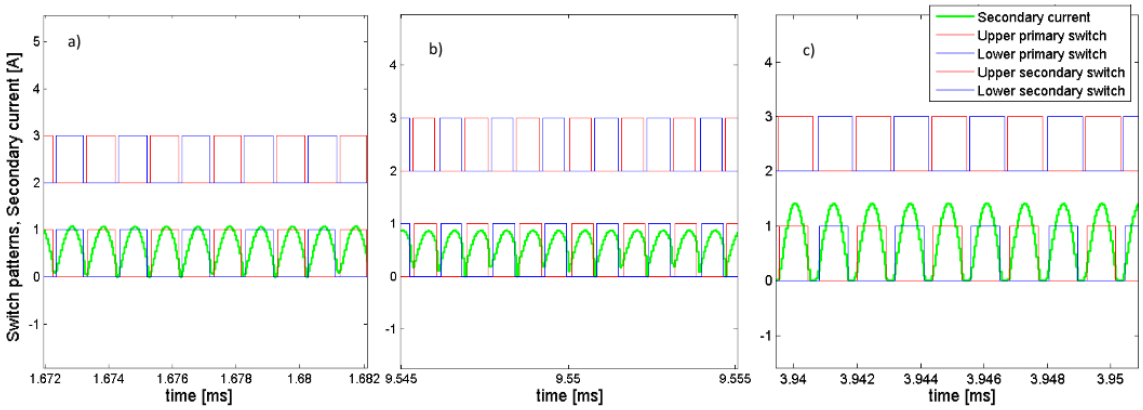

Figure 27: Synchronous rectifier gate pulses a) at resonant frequency b) above resonant frequency c) below resonant frequency.

In Figure 27b., the converter is operated above the resonant frequency. Now ZCS conditions are no longer possible, and the increased switching losses can be easily observed. Especially during the turn-offs, switching takes place while the current flowing through the switches is still fairly large. However, the secondary synchronization algorithm seems to work fairly well, as the secondary pulses are appropriately delayed in relation to the primary pulses.

In Figure 27c., the converter is operated below the resonant frequency. Higher resonant tank gain can be seen as larger current amplitude in the secondary. As the primary side pulse durations are now longer than the resonant period, the switches in the secondary side must have some non-conducting time in between the current pulses. The duty cycle of the secondary switches is correctly limited to the resonant frequency duty cycle, and the pulses are properly advanced in relation to the primary pulses. Zero-current switching is achieved.

4.5 PWM Module

Figure 28 is a screenshot taken from one of the VHDL simulations. It shows the control pulses of the primary side switches as well as the related duty, period and synchronization command words from the controller. The command words are actually binary words with widths of 14 bits, but for convenience their radix has been set to unsigned fixed point format in the simulator. As is shown in the figure, the pulses are properly out of phase, with some dead time in between to avoid shoot throughs. The synchronization command of 1.5 means that when the counter of the first channel reaches its middle point, the counter of the second channel should be at the value of 1.5. However, the counters can only assume integer values, so the fractional part is achieved by advancing or delaying the rising and falling edges of the pulses.

The duty command for both channels in the figure is set to 143.5. In nanoseconds,

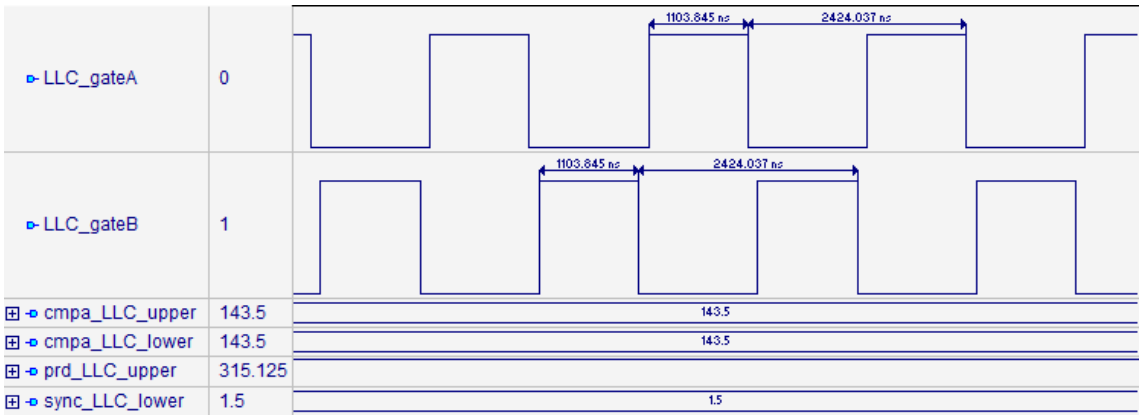

Figure 28: Primary half-bridge PWM channels

with PWM clock rate of 130 MHz, this corresponds to:

$$Pulsewidth = 143.5 \cdot \frac{1}{130 \cdot 10^6 Hz} = 1103.85ns. \tag{50}$$

The period is always the same for all the channels within one PWM module. In this case, the period command is set to 315.125, which corresponds to:

$$Period = 315.125 \cdot \frac{1}{130 \cdot 10^6 Hz} = 2424.04ns. \tag{51}$$

As can be seen in Figure 28, the PWM module outputs correspond to the command words.

4.6 Floating-Point Arithmetic

In the following simulation screenshots, the floating-point subtraction and multiplication is demonstrated. The radix of the binary words is chosen as floating-point in the simulator interface for more convenient viewing.

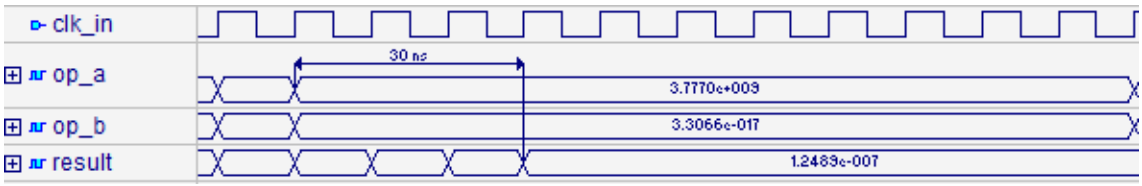

Figure 29: Floating-point multiplication

in Figure 29, the floating-point multiplication is shown. After the operands *op_a* and *op_b* have settled, signal *result* settles after 3 clock periods, corresponding to the 3-stage pipeline implementation described in section 3.5.4. in Figure 30, the floating-point subtraction is shown. 50 ns or five input clock periods after operands have settled, the result settles.

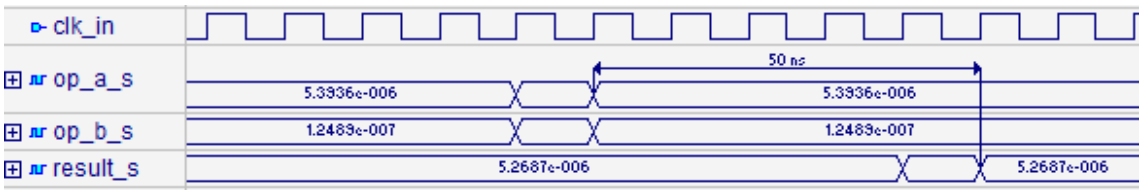

Figure 30: Floating-point subtraction

4.7 Controller Delay

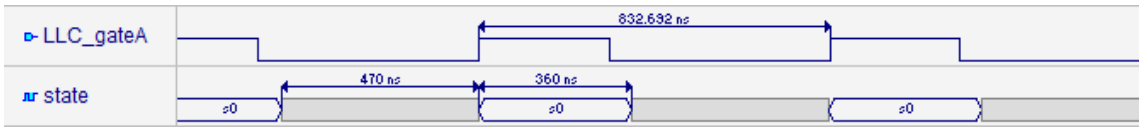

Figure 31: Control computation at maximum control rate

in Figure 31, the LLC converter is run at the maximum frequency of about 1.2 MHz. Below the upper primary switch signal, signal *state* is shown. Signal *state* keeps track of the states, or steps of the control algorithm. When the control computation is not active, the state machine is idly waiting in the first state, *s0*. As the PWM counter reaches the point where the sampling is set to take place, the analog-to-digital conversions are performed and the control is triggered upon the completion of the conversions. State signal then begins running through all the states, during which the control algorithm is computed and all the controlled variables are assigned new values based on the past values and the current measurements. The control computation takes 470 ns to complete, leaving at least 360 ns of idle time for each switching cycle.

From the previous observations, it can be determined that an ADC with sampling speed of at least 2.8 MSPS is required for one cycle control with maximum control rate of 1.2 MHz. Since the control algorithm begins with the output voltage control loop, the current sample is not required to be ready at the very beginning of the control computation and its sampling can take somewhat longer. For the primary current, an ADC with a speed of at least 2 MSPS is required.

5 Conclusions

The conventional solution for real-time digital control of switched-mode power supplies has been high-performance microcontrollers. Their inherent limitation is the finite processing time available for tasks. The clock frequency of the microcontroller places a strict limit to the maximum complexity and execution rate of the control algorithm. Demand for higher power density and higher efficiency together with the new wide band-gap devices such as GaN switches has grown interest in higher control rates and switching frequencies. As the FPGA technology offers parallel computation, its capacity for complex high-frequency control is far greater than that of microcontrollers. The limiting factor has been the relatively high cost of FPGA chips and the necessary peripheral devices.

The goal of this thesis was to assess the feasibility of implementing complex control for a telecom AC-DC power supply with a low-cost FPGA. To reach this goal and gain an overview of the issues that should be accounted for, a resonant LLC converter controller was implemented. The controller includes the control algorithm itself as well as the necessary PWM and arithmetic modules. First, the best way to partition both the physical and the functional side of the control was researched. Several different control partitioning options were compared to each other. Factors taken into account were cost, control performance, firmware upgradability, electrical surge robustness and development effort considerations. Placing the FPGA chip on the secondary side of the galvanic isolation and a low-cost DSP on the primary side offered the best balance of mentioned factors. The advantages included lowest cost, best protection against surge voltages and the possibility to raise the switching frequency even higher in the future without requiring repartitioning the control system. The main disadvantages of this partitioning were the requirement for an internal control bus and the difficulty of updating the software and logic configuration of the controllers in the field.

A synchronous high-resolution PWM module was implemented and simulated. While most low-cost microcontrollers do not even have high-resolution PWM, on an FPGA the modulator can be built directly from the needs of the application. In this case, multiple channels had to be synchronizable to each other, and the frequency had to be variable in high-resolution. Implementing such functionality by manipulating the PWM registers of a microcontroller is challenging.

The secondary synchronization was analyzed using the first harmonic approximation. The required dynamic range implied the need for floating-point arithmetic. As the requirements for the control computation delay was strict, custom floating-point modules were implemented. These modules were designed for optimum execution rate and delay to satisfy the needs of the control. The execution rate, delay and resource consumption were balanced with a pipelined design. Available hardware blocks were utilized to maximize the speed of the calculations. In addition to the necessary number format conversions, the implemented modules were subtraction, multiplication and multiplicative inverse.

Finally a complete VHDL simulation model was created and the individual modules were connected into a working control system. The functionality of the

control was verified by running several simulations with different parameters such as the load and the reference output voltage. The frequency modulation control of the LLC primary half-bridge and the control of the synchronous rectifier were found to operate well in varying conditions.

The next step would be to study the ways of implementing high-performance analog-to-digital converters with resources available on low-cost FPGAs. If the control measurements could be scaled appropriately, it might be possible to use available high-speed, low-resolution FPGA ADC architectures for the measurements. This would remove the need for external ADCs and reduce the control system cost considerably. Additionally, it could be feasible to integrate the microcontroller into the FPGA as well, using some of the available soft-core architectures. This would also reduce the total cost of the control system. Together with low switching loss devices, FPGA controllers allow the development of much smaller and efficient power supplies that help fulfill the growing power demands of the telecommunication networks.

References

[1] Steigerwald, R. L. High-Frequency Resonant Transistor DC-DC Converters. *IEEE Trans. on Industrial Electronics,* vol. 31, no. 2, pp. 1175–1204, 1984.

[2] Tötterman, M. and Grigore, V. Digital Control Partitioning for Telecom AC-DC Power Supply. *in Proc. IEEE 34th Intl. Telecommunications Energy Conf. (INTELEC),* pp. 1–6, 2012.

[3] IEC 61000-3-2. Electromagnetic compatibility (EMC) - Part 3-2: Limits. 4th ed. International Electrotechnical Commission, 2014. 69 pages.

[4] Rashid, M. H. *Power Electronics Handbook.* 3rd ed., Elsevier Science, 2011, ISBN 978-0-123-82037-2.

[5] Huber, L. et. al. Performance Evaluation of Bridgeless PFC Boost Rectifiers. *IEEE Trans. on Power Electronics,* vol. 23, no. 3, pp. 1381–1390, 2008.

[6] Balogh, L. and Redl, R. Power-Factor Correction with Interleaved Boost Converters in Continuous-Inductor-Current Mode. *in Proc. IEEE 8th Appl. Power Electronics Conf. (APEC),* pp. 168–174, 1993.

[7] Choudhury, S. and Noon, J. P. A DSP based Digitally Controlled Interleaved PFC Converter *in Proc. IEEE 8th Appl. Power Electronics Conf. (APEC),* pp. 648–654, 2005.

[8] Yang, B. et. al. LLC Resonant Converter for Front End DC/DC Conversion. *in Proc. IEEE 17th Appl. Power Electronics Conf. (APEC),* pp. 1108–1112, 2002.

[9] STMicroelectronics. Application note AN2644. An introduction to LLC resonant half-bridge converter. 2008. Available in URL `http://www.st.com/content/ccc/resource/technical/document/application_note/de/f9/17/b7/ad/9f/4d/dd/CD00174208.pdf/files/CD00174208.pdf/jcr:content/translations/en.CD00174208.pdf` on 21.6.2017.

[10] Mohan, N. *Power Electronics. A First Course.* John Wiley & Sons, Inc., 2012, ISBN 978-1-118-07480-0.

[11] Ghahderijani, M. M. et. al. Frequency-Modulation Control of a DC/DC Current-Source Parallel-Resonant Converter. *IEEE Trans. on Industrial Electronics,* vol. 64, no. 7, pp. 5392–5402, 2017.

[12] Hauck, S. and DeHon, A. *Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation.* 1st ed., Elsevier Science, 2007, ISBN 978-0-080-55601-7.

[13] Monmasson, E. and Cirstea, M. N. FPGA Design Methodology for Industrial Control Systems - A Review. *IEEE Trans. on Industrial Electronics,* vol. 54, no. 4, pp. 1824–1842, 2007.

[14] García-Tenori, J. M. Digital Control Techniques for DC/DC Power Converters. Online Document. Master's Thesis, Helsinki University of Technology, Helsinki, Finland, 2009. Available in URL `https://aaltodoc.aalto.fi/bitstream/123456789/3111/1/urn100074.pdf` on 21.6.2017.

[15] Ramsden, E. *Hall-Effect Sensors - Theory and Application.* 2nd ed., Elsevier Science, 2006, ISBN 978-0-0805-2374-3.

[16] Peterchev, A. V. and Sanders, S. R. Quantization Resolution and Limit Cycling in Digitally Controlled PWM Converters. *IEEE Trans. on Power Electronics,* vol. 18, no. 1, pp. 301–308, 2003.

[17] Navarro, D. et. al. Synchronous FPGA-based High-Resolution Implementations of Digital Pulse-Width Modulators. *IEEE Trans. on Power Electronics,* vol. 27, no. 5, pp. 2515–2525, 2012.

[18] Santa, C. H. et. al. FPGA based Digital Pulse Width Modulator with Time Resolution under 2 ns. *in Proc. IEEE 22nd Appl. Power Electronics Conf. (APEC),* pp. 877–881, 2007.

[19] Batarseh, M. G. et. al. Segmented Digital Clock Manager - FPGA based Digital Pulse Width Modulator Technique. *in Proc. IEEE Power Electronics Specialists Conf. (PESC),* pp. 3036–3042, 2008.

[20] Castro, A. de and Todorovich, E. DPWM vased on FPGA Clock Phase Shifting with Time Resolution under 100 ps. *in Proc. IEEE Power Electronics Specialists Conf. (PESC),* pp. 3054–3059, 2008.

[21] Quintero, J. et. al. FPGA based Digital Control with High-Resolution Synchronous DPWM and High-Speed Embedded A/D Converter. *in Proc. IEEE 24th Appl. Power Electronics Conf. (APEC),* pp. 1360–1366, 2009.

[22] Lattice Semiconductor. Datasheet. ECP5 and ECP5-5G Family, 2017. Available in URL `http://www.latticesemi.com/view_document?document_id=50461` on 21.6.2017.

[23] Lattice Semiconductor. Reference Manual. LatticeMico32 Processor, 2012, version 3.9. Available in URL `http://www.latticesemi.com/view_document?document_id=52077` on 21.6.2017.

[24] IEEE 754-2008. IEEE Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, 2008. 58 pages.

[25] Ligon, W. B. et. al. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. *in Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FPGA),* pp. 206–215, 1998.

[26] Fagin, B. and Renard, C. Field Programmable Gate Arrays and Floating Point Arithmetic. *IEEE Trans. on VLSI Syst.* vol. 2, no. 3, pp. 365–367, 1994.

[27] Govindu, G. et. al. Analysis of High-performance Floating-point Arithmetic on FPGAs. *in Proc. 18th Intl. Parallel and Distributed Processing Symp. (IPDPS),* pp. 149–156, 2004.

[28] Oberman, S. F. et. al. The SNAP Project: Design of Floating Point Arithmetic Units. *in Proc. IEEE 13th Symp. on Computer Arithmetic. (ARITH),* pp. 156–165, 1997.

A PWM module

In this appendix, the VHDL code for all of the submodules of the PWM implementation are included as listings.

A.1 Top-Level

```
-- This is the top PWM module.
-- clk_in is the input clock
-- dcycle_in_x are the duty cycle commands for x channels
-- adctime_in is the adc synchronization command
-- sync_x is the synchronization command for other channels than 1.
-- sforce_chx is the force_off signal for x channels
-- rst_n is the active-low reset
-- dfreq_in is the commond pwm period command for all channels
-- control_done is an input flag informing that the control computation is finished
-- adc_flag_out is an output flag informing that the AD conversion is ready
-- dpwm and dpwm_chx are the pwm outputs
-- Written by Tero Kuusijarvi, 27.7.2017

library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_arith.all;
use ieee. std_logic_unsigned.all;
library lattice;

entity pwm_top is
generic ( n_channels : integer := 4); -- must be equal or larger than 2
port(
  clk_in : in std_logic;
  dcycle_in :   in std_logic_vector (13 downto 0);
  dfreq_in :    in std_logic_vector (13 downto 0);
  adctime_in :  in std_logic_vector (10 downto 0);
  dcycle_in_2 :   in std_logic_vector (13 downto 0);
  sync_2 :    in std_logic_vector (13 downto 0);
  dcycle_in_3 :   in std_logic_vector (13 downto 0);
  sync_3 :    in std_logic_vector (13 downto 0);
  dcycle_in_4 :   in std_logic_vector (13 downto 0);
  sync_4 :    in std_logic_vector (13 downto 0);
  sforce_ch1 :  in std_logic;
  sforce_ch2 :  in std_logic;
  sforce_ch3 :  in std_logic;
  sforce_ch4 :  in std_logic;
  rst_n :     in std_logic;
  dpwm :      out std_logic;
  dpwm_ch2 :    out std_logic;
  dpwm_ch3 :    out std_logic;
  dpwm_ch4 :    out std_logic;
  adc_flag_out :    out std_logic;
  control_done : in std_logic
  );
end pwm_top;

architecture rtl of pwm_top is

component srlatch -- asynchronous set-reset
port (s_in,r_in: in  std_logic;
  q_out: out  std_logic);
end component;

component pll_ecp551 -- pll module
port (clki: in  std_logic; clkop: out  std_logic;
  clkos: out  std_logic; clkos2: out  std_logic;
  clkos3: out  std_logic);
end component;
```

```
59
60
61 component counter is -- duty and sync and sforces for channels 2 to n must be changed if
       number of channels is changed
62 generic ( n_channels : integer := 4);
63 port (
64   clock : in  std_logic;
65   aclr : in  std_logic;
66   freq : in std_logic_vector(13 downto 0);
67   duty : in std_logic_vector(13 downto 0);
68   adctime : in std_logic_vector(10 downto 0);
69   duty_ch2 : in std_logic_vector(13 downto 0);
70   sync_ch2 : in std_logic_vector(13 downto 0);
71   duty_ch3 : in std_logic_vector(13 downto 0);
72   sync_ch3 : in std_logic_vector(13 downto 0);
73   duty_ch4 : in std_logic_vector(13 downto 0);
74   sync_ch4 : in std_logic_vector(13 downto 0);
75   sforce_ch1 :  in std_logic;
76   sforce_ch2 :  in std_logic;
77   sforce_ch3 :  in std_logic;
78   sforce_ch4 :  in std_logic;
79   setd : out std_logic;
80   clrd : out std_logic;
81   adc_flag : out std_logic;
82   setd_ch : out std_logic_vector(2 to n_channels);
83   clrd_ch : out std_logic_vector(2 to n_channels);
84   dpwm_set_lsbs : out  std_logic_vector(2 downto 0);
85   dpwm_reset_lsbs : out  std_logic_vector(2 downto 0);
86   dpwm_set_lsbs_ch : out  std_logic_vector((3*n_channels - 4) downto 0);
87   dpwm_reset_lsbs_ch : out  std_logic_vector((3*n_channels - 4) downto 0));
88 end component;
89
90 component mux is
91 port (
92   sel : in std_logic_vector(2 downto 0);
93   in0 : in std_logic;
94   in1 : in std_logic;
95   in2 : in std_logic;
96   in3 : in std_logic;
97   in4 : in std_logic;
98   in5 : in std_logic;
99   in6 : in std_logic;
100  in7 : in std_logic;
101  output : out std_logic);
102 end component;
103
104 signal clock_0 : std_logic;
105 signal clock_45 : std_logic;
106 signal clock_90 : std_logic;
107 signal clock_135 : std_logic;
108
109 signal adc_flag : std_logic;
110
111 signal reset_lsbs, set_lsbs : std_logic_vector(2 downto 0);
112 signal set, reset : std_logic;
113 signal setd : std_logic;
114 signal clrd : std_logic;
115 signal dcycle_reg, dfreq_reg : std_logic_vector (13 downto 0) ;
116 signal adctime_reg : std_logic_vector(10 downto 0);
117
118 type ffabc is array (0 to 3) of std_logic;
119 signal ffabcd : ffabc;
120 type ff0123 is array (0 to 7) of std_logic;
121 signal ff01234567 : ff0123;
122 signal set_ff01234567 : ff0123;
123
124 -- the following signals must be changed if number of channels is changed
125 signal reset_lsbs_ch, set_lsbs_ch : std_logic_vector(3*n_channels - 4 downto 0);
```

```
126
127
128 type asynch_sr_type is array (2 to n_channels) of std_logic;
129 signal set_ch, reset_ch : asynch_sr_type;
130 signal setd_ch, clrd_ch : std_logic_vector(2 to n_channels);
131 type dcycle_reg_type is array (2 to n_channels) of std_logic_vector(13 downto 0);
132 signal dcycle_reg_ch, sync_reg_ch : dcycle_reg_type;
133
134 type ffabc_ch is array (2 to n_channels) of ffabc;
135 signal ffabcd_ch : ffabc_ch;
136
137 type ff0123_ch is array (2 to n_channels) of std_logic_vector(0 to 1);
138 signal ff04_ch, ff15_ch, ff26_ch, ff37_ch : ff0123_ch;
139 signal set_ff04_ch, set_ff15_ch, set_ff26_ch, set_ff37_ch : ff0123_ch;
140
141 type output_ch is array (2 to n_channels) of std_logic;
142 signal dpwm_ch : output_ch;
143
144 type sforce_type is array (1 to n_channels) of std_logic;
145 signal sforce_arr : sforce_type;
146
147 begin
148 --these must be reconfigured when number of channels is changed.
149 dpwm_ch2 <= dpwm_ch(2);
150 dpwm_ch3 <= dpwm_ch(3);
151 dpwm_ch4 <= dpwm_ch(4);
152
153 pll_exa_inst : pll_ecp551
154 port map (
155   clki  =>clk_in,
156   clkop =>clock_0,
157   clkos =>clock_45,
158   clkos2  =>clock_90,
159   clkos3  =>clock_135
160   );
161
162 srff_inst : srlatch
163 port map (
164   s_in  => set,
165   r_in  => reset,
166   q_out => dpwm
167   );
168
169 mux_inst_rst : mux
170 port map(
171   sel => reset_lsbs,
172   in0 => ff01234567(0),
173   in1 => ff01234567(1),
174   in2 => ff01234567(2),
175   in3 => ff01234567(3),
176   in4 => ff01234567(4),
177   in5 => ff01234567(5),
178   in6 => ff01234567(6),
179   in7 => ff01234567(7),
180   output => reset);
181
182 mux_inst_set : mux
183 port map(
184   sel => set_lsbs,
185   in0 => set_ff01234567(0),
186   in1 => set_ff01234567(1),
187   in2 => set_ff01234567(2),
188   in3 => set_ff01234567(3),
189   in4 => set_ff01234567(4),
190   in5 => set_ff01234567(5),
191   in6 => set_ff01234567(6),
192   in7 => set_ff01234567(7),
193   output => set );
```

```
194
195 count_inst : counter -- duty and sync for channels 2 to n must be reconfigured if number
        of channels is changed.
196 generic map ( n_channels => n_channels )
197 port map (
198   clock => clock_0,
199   aclr => rst_n,
200   freq => dfreq_reg,
201   duty => dcycle_reg,
202   adctime => adctime_reg,
203   duty_ch2 => dcycle_reg_ch(2),
204   sync_ch2 => sync_reg_ch(2),
205   duty_ch3 => dcycle_reg_ch(3),
206   sync_ch3 => sync_reg_ch(3),
207   duty_ch4 => dcycle_reg_ch(4),
208   sync_ch4 => sync_reg_ch(4),
209   sforce_ch1 => sforce_arr(1),
210   sforce_ch2 => sforce_arr(2),
211   sforce_ch3 => sforce_arr(3),
212   sforce_ch4 => sforce_arr(4),
213   setd => setd,
214   clrd => clrd,
215   adc_flag => adc_flag,
216   setd_ch => setd_ch,
217   clrd_ch => clrd_ch,
218   dpwm_set_lsbs => set_lsbs,
219   dpwm_reset_lsbs => reset_lsbs,
220   dpwm_set_lsbs_ch => set_lsbs_ch,
221   dpwm_reset_lsbs_ch => reset_lsbs_ch );
222
223 gen_comp_ch : for n in 2 to n_channels generate
224   srff_inst_ch2 : srlatch
225   port map (
226     s_in  => set_ch(n),
227     r_in  => reset_ch(n),
228     q_out => dpwm_ch(n) );
229
230   mux_inst_rst_ch : mux
231   port map(
232     sel => reset_lsbs_ch(3*n - 4 downto 3*n - 6),
233     in0 => ff04_ch(n)(0),
234     in1 => ff15_ch(n)(0),
235     in2 => ff26_ch(n)(0),
236     in3 => ff37_ch(n)(0),
237     in4 => ff04_ch(n)(1),
238     in5 => ff15_ch(n)(1),
239     in6 => ff26_ch(n)(1),
240     in7 => ff37_ch(n)(1),
241     output => reset_ch(n) );
242
243   mux_inst_set_ch : mux
244   port map(
245     sel => set_lsbs_ch(3*n - 4 downto 3*n - 6),
246     in0 => set_ff04_ch(n)(0),
247     in1 => set_ff15_ch(n)(0),
248     in2 => set_ff26_ch(n)(0),
249     in3 => set_ff37_ch(n)(0),
250     in4 => set_ff04_ch(n)(1),
251     in5 => set_ff15_ch(n)(1),
252     in6 => set_ff26_ch(n)(1),
253     in7 => set_ff37_ch(n)(1),
254     output => set_ch(n));
255 end generate;
256
257
258 process(clock_0, rst_n) -- input register for duty, frequency
259 begin
260   if (rst_n = '0') then
```

```
261     dcycle_reg <= "00100100011000";
262     dfreq_reg <= "01010001010000";
263     adctime_reg <= "00101111001";
264     adc_flag_out <= '0';
265     sforce_arr(1) <= '0';
266     for i in 2 to n_channels loop
267       dcycle_reg_ch(i) <= "00100100011000";
268       sync_reg_ch(i) <= "01010001001000";
269       sforce_arr(i) <= '0';
270     end loop;
271   elsif(rising_edge(clock_0)) then  -- when cnt >= dfreq_reg/2, registers get updated
272     if control_done = '1' then
273       dcycle_reg <= dcycle_in;
274       dfreq_reg <= dfreq_in;
275       adctime_reg <= adctime_in;
276       -- following  must be changed if number of channels is changed.
277       dcycle_reg_ch(2) <= dcycle_in_2;
278       sync_reg_ch(2) <= sync_2;
279       dcycle_reg_ch(3) <= dcycle_in_3;
280       sync_reg_ch(3) <= sync_3;
281       dcycle_reg_ch(4) <= dcycle_in_4;
282       sync_reg_ch(4) <= sync_4;
283       sforce_arr(1) <= sforce_ch1;
284       sforce_arr(2) <= sforce_ch2;
285       sforce_arr(3) <= sforce_ch3;
286       sforce_arr(4) <= sforce_ch4;
287     end if;
288     adc_flag_out <= adc_flag;
289   end if;
290 end process;
291
292
293 process(clock_0, rst_n) -- registers clocked to clock_0
294 begin
295   if (rst_n = '0') then
296     ffabcd <= (others => '0');
297     ff01234567(0) <= '0';
298     ff01234567(4) <= '0';
299     set_ff01234567(0) <= '0';
300     set_ff01234567(4) <= '0';
301
302     for i in 2 to n_channels loop
303       ffabcd_ch(i) <= (others => '0');
304       ff04_ch(i) <= (others => '0');
305       set_ff04_ch(i) <= (others => '0');
306     end loop;
307
308   elsif(rising_edge(clock_0)) then
309     ffabcd <= (setd, clrd, ffabcd(0), ffabcd(1));
310     ff01234567(4) <= ff01234567(0);
311     set_ff01234567(4) <= set_ff01234567(0);
312
313     for i in 2 to n_channels loop
314       ffabcd_ch(i) <= (setd_ch(i), clrd_ch(i), ffabcd_ch(i)(0), ffabcd_ch(i)(1));
315       ff04_ch(i)(1) <= ff04_ch(i)(0);
316       set_ff04_ch(i)(1) <= set_ff04_ch(i)(0);
317     end loop;
318
319   elsif(falling_edge(clock_0)) then
320     ff01234567(0) <= ffabcd(3);
321     set_ff01234567(0) <= ffabcd(2);
322
323     for i in 2 to n_channels loop
324       ff04_ch(i)(0) <= ffabcd_ch(i)(3);
325       set_ff04_ch(i)(0) <= ffabcd_ch(i)(2);
326     end loop;
327   end if;
328 end process;
```

```
329
330
331 process(clock_45, rst_n)
332 begin
333   if (rst_n = '0') then
334     ff01234567(1) <= '0';
335     ff01234567(5) <= '0';
336     set_ff01234567(1) <= '0';
337     set_ff01234567(5) <= '0';
338
339     for i in 2 to n_channels loop
340       ff15_ch(i) <= (others => '0');
341       set_ff15_ch(i) <= (others => '0');
342     end loop;
343
344   elsif(rising_edge(clock_45)) then
345     ff01234567(5) <= ff01234567(1);
346     set_ff01234567(5) <= set_ff01234567(1);
347
348     for i in 2 to n_channels loop
349       ff15_ch(i)(1) <= ff15_ch(i)(0);
350       set_ff15_ch(i)(1) <= set_ff15_ch(i)(0);
351     end loop;
352
353   elsif(falling_edge(clock_45)) then
354     ff01234567(1) <= ffabcd(3);
355     set_ff01234567(1) <= ffabcd(2);
356
357     for i in 2 to n_channels loop
358       ff15_ch(i)(0) <= ffabcd_ch(i)(3);
359       set_ff15_ch(i)(0) <= ffabcd_ch(i)(2);
360     end loop;
361
362   end if;
363 end process;
364
365 process(clock_90, rst_n)
366 begin
367   if (rst_n = '0') then
368     ff01234567(2) <= '0';
369     ff01234567(6) <= '0';
370     set_ff01234567(2) <= '0';
371     set_ff01234567(6) <= '0';
372
373   for i in 2 to n_channels loop
374     ff26_ch(i) <= (others => '0');
375     set_ff26_ch(i) <= (others => '0');
376   end loop;
377
378   elsif(rising_edge(clock_90)) then
379     ff01234567(6) <= ff01234567(2);
380     set_ff01234567(6) <= set_ff01234567(2);
381
382     for i in 2 to n_channels loop
383       ff26_ch(i)(1) <= ff26_ch(i)(0);
384       set_ff26_ch(i)(1) <= set_ff26_ch(i)(0);
385     end loop;
386
387   elsif(falling_edge(clock_90)) then
388     ff01234567(2) <= ffabcd(3);
389     set_ff01234567(2) <= ffabcd(2);
390
391     for i in 2 to n_channels loop
392       ff26_ch(i)(0) <= ffabcd_ch(i)(3);
393       set_ff26_ch(i)(0) <= ffabcd_ch(i)(2);
394     end loop;
395
396   end if;
```

```
397 end process;
398
399
400 process(clock_135, rst_n)
401 begin
402   if (rst_n = '0') then
403     ff01234567(3) <= '0';
404     ff01234567(7) <= '0';
405     set_ff01234567(3) <= '0';
406     set_ff01234567(7) <= '0';
407
408   for i in 2 to n_channels loop
409     ff37_ch(i) <= (others => '0');
410     set_ff37_ch(i) <= (others => '0');
411   end loop;
412
413   elsif(rising_edge(clock_135)) then
414     ff01234567(7) <= ff01234567(3);
415     set_ff01234567(7) <= set_ff01234567(3);
416
417     for i in 2 to n_channels loop
418       ff37_ch(i)(1) <= ff37_ch(i)(0);
419       set_ff37_ch(i)(1) <= set_ff37_ch(i)(0);
420     end loop;
421
422   elsif(falling_edge(clock_135)) then
423     ff01234567(3) <= ffabcd(3);
424     set_ff01234567(3) <= ffabcd(2);
425
426     for i in 2 to n_channels loop
427       ff37_ch(i)(0) <= ffabcd_ch(i)(3);
428       set_ff37_ch(i)(0) <= ffabcd_ch(i)(2);
429     end loop;
430
431   end if;
432 end process;
433
434 end rtl;
```

A.2 Counter

```
-- This is the PWM counter module.
-- Written by Tero Kuusijarvi 05.04.2017

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity counter is
generic( n_channels : integer := 4);
port(
  clock : in  std_logic;
  aclr : in  std_logic;
  freq : in std_logic_vector(13 downto 0);
  duty : in std_logic_vector(13 downto 0);
  adctime : in std_logic_vector(10 downto 0);
  duty_ch2 : in std_logic_vector(13 downto 0);
  sync_ch2 : in std_logic_vector(13 downto 0);
  duty_ch3 : in std_logic_vector(13 downto 0);
  sync_ch3 : in std_logic_vector(13 downto 0);
  duty_ch4 : in std_logic_vector(13 downto 0);
  sync_ch4 : in std_logic_vector(13 downto 0);
  sforce_ch1 :  in std_logic;
  sforce_ch2 :  in std_logic;
  sforce_ch3 :  in std_logic;
  sforce_ch4 :  in std_logic;
  setd : out std_logic;
  clrd : out std_logic;
  adc_flag : out std_logic;
  setd_ch : out std_logic_vector(2 to n_channels);
  clrd_ch : out std_logic_vector(2 to n_channels);
  dpwm_set_lsbs : out  std_logic_vector(2 downto 0);
  dpwm_reset_lsbs : out  std_logic_vector(2 downto 0);
  dpwm_set_lsbs_ch : out  std_logic_vector((3*n_channels - 4) downto 0);
  dpwm_reset_lsbs_ch : out  std_logic_vector((3*n_channels - 4) downto 0) );
end counter;

architecture rtl of counter is
signal counter_sig : unsigned(10 downto 0);
signal freq_in, duty_in : std_logic_vector(13 downto 0);
signal count : unsigned(10 downto 0);
signal mpt : unsigned(10 downto 0);
signal mp0 : unsigned(13 downto 0);
signal dpwm_set : unsigned(13 downto 0);
signal dpwm_reset : unsigned(13 downto 0);
signal mpt_passed : std_logic;

type cntr_sig_type is array (2 to n_channels) of unsigned(10 downto 0);
signal counter_sig_arr : cntr_sig_type;
type duty_in_type is array (2 to n_channels) of std_logic_vector(13 downto 0);
signal duty_in_arr : duty_in_type;
type sync_in_type is array (2 to n_channels) of std_logic_vector(13 downto 0);
signal sync_in_arr : sync_in_type;
type duty_saved_type is array (2 to n_channels) of unsigned(13 downto 0);
signal duty_saved_arr : duty_saved_type;
type count_type is array (2 to n_channels) of unsigned(10 downto 0);
signal count_arr : count_type;
type mpt_type is array (2 to n_channels) of unsigned(10 downto 0);
signal mpt_arr : mpt_type;
type mp0_type is array (2 to n_channels) of unsigned(13 downto 0);
signal mp0_arr : mp0_type;
type dpwm_set_type is array (2 to n_channels) of unsigned(13 downto 0);
signal dpwm_set_arr : dpwm_set_type;
type dpwm_reset_type is array (2 to n_channels) of unsigned(13 downto 0);
signal dpwm_reset_arr : dpwm_reset_type;
type mpt_passed_type is array (2 to n_channels) of std_logic;
```

```
67  signal mpt_passed_arr : mpt_passed_type;
68  type sync_glitch_type is array (2 to n_channels) of unsigned(0 downto 0);
69  signal sync_glitch_arr : sync_glitch_type;
70  type reset_calctd_type is array (2 to n_channels) of std_logic;
71  signal reset_calctd_arr : reset_calctd_type;
72
73  type onezero_type is array (1 to n_channels) of std_logic;
74  signal duty_one_arr : onezero_type;
75  signal duty_zero_arr : onezero_type;
76  signal sforce_arr : onezero_type;
77
78  type reset_comp_type is array (2 to n_channels) of std_logic;
79  signal rst_comp_arr : reset_comp_type;
80  signal set_comp_arr : reset_comp_type;
81
82  begin
83    process(clock, aclr)
84    variable counter_temp : unsigned(11 downto 0);
85    variable mpt_temp : unsigned(13 downto 0);
86    variable count_var : unsigned(10 downto 0);
87    variable mp0_temp : unsigned(13 downto 0);
88    begin
89      if(aclr = '0') then -- initial values
90        counter_sig <= "00000000001";
91        mpt_passed <= '0';
92        mp0 <= "001010001010000";
93        mpt <= "001010001010";
94        count <= "010100010100";
95
96        dpwm_set <= (others => '1');
97        dpwm_reset <= (others => '0');
98
99        for i in 2 to n_channels loop
100         count_arr(i) <= "010100010100";
101         counter_sig_arr(i) <= "00000000001";
102         dpwm_set_arr(i) <= (others => '1');
103         duty_saved_arr(i) <= "001001000110000";
104         mpt_arr(i) <= "001010001010";
105         mp0_arr(i) <= "001010001010000";
106         sync_glitch_arr(i) <= "0";
107       end loop;
108
109     elsif(rising_edge(clock)) then
110       if(counter_sig >= count) then -- resetting the counter
111         counter_sig <= "00000000001";
112         dpwm_set <= resize(('0' & mp0) +
113           ('0' & not('0' & unsigned(duty_in(13 downto 1)))) +
114           (to_unsigned(0, 14) & not(duty_in(0))), 14);
115
116         mpt_passed <= '0';
117         for i in 2 to n_channels loop
118           dpwm_set_arr(i) <= resize(('0' & mp0_arr(i)) +
119             ('0' & not('0' & unsigned((duty_saved_arr(i)(13 downto 1))))) +
120             (to_unsigned(0, 14) & not((duty_saved_arr(i)(0)))) -
121             (unsigned(sync_in_arr(i)(2 downto 0))), 14);
122
123           if(counter_sig_arr(i) >= count_arr(i) or
124             (counter_sig_arr(i) = to_unsigned(0, 11))) then
125
126             counter_sig_arr(i) <= "00000000001";
127           else
128             counter_temp := ('0' & counter_sig_arr(i)) + to_unsigned(1, 12);
129             counter_sig_arr(i) <= counter_temp(10 downto 0);
130           end if;
131         end loop;
132       elsif(counter_sig >= mpt and not(mpt_passed = '1')) then -- crossing the middle
      point of the carrier period
133         for i in 2 to n_channels loop
```

```
134           duty_saved_arr(i) <= unsigned(duty_in_arr(i));
135           count_var := resize((’0’ & mpt_arr(i)) +
136             ("00" & unsigned(freq_in(13 downto 4))), 11);
137
138           if(counter_sig_arr(i) = unsigned(sync_in_arr(i)(13 downto 3)) and
139             (((counter_sig_arr(i)(10 downto 0) >= dpwm_set_arr(i)(13 downto 3)))) and
140             (counter_sig_arr(i) < dpwm_reset_arr(i)(13 downto 3))) then
141
142             sync_glitch_arr(i) <= "1";
143           else
144             sync_glitch_arr(i) <= "0";
145           end if;
146
147           if(mp0(2 downto 0) = "000" and
148             ((counter_sig_arr(i)(10 downto 0) >= dpwm_set_arr(i)(13 downto 3))))
149             then -- synchronization glitch would happen here if not accounted for
150             -- glitch: counter_sig_ch2 keeps its value for 2 consecutive clock cycles
151             -- also count_ch2 is increased while setd_ch2 is already set
152             -- the next setd_ch2 set happens one cycle too late
153             -- with counter_sig_ch2 < dpwm_set_ch2, theres no glitch.
154             count_arr(i) <= resize(((’0’ & count_var) + ("011111111111")), 11);
155           else
156             count_arr(i) <= count_var;
157           end if;
158           counter_sig_arr(i) <= unsigned(sync_in_arr(i)(13 downto 3));
159
160           mp0_temp := resize(((’0’ & mp0_arr(i)) + (’0’ & unsigned(freq_in)) +
161             (’0’ & not(count_var & "000")) + to_unsigned(1, 15)), 14);
162
163           mp0_arr(i) <= (mp0_temp);
164           mpt_arr(i) <= mp0_temp(13 downto 3);
165         end loop;
166
167         count_var := resize((’0’ & mpt) + ("00" & unsigned(freq_in(13 downto 4))), 11);
168         count <= count_var;
169
170         mp0_temp := resize(((’0’ & mp0) + (’0’ & unsigned(freq_in)) +
171           (’0’ & not(count_var & "000")) + to_unsigned(1, 15)), 14);
172
173         mp0 <= mp0_temp;
174         mpt <= mp0_temp(13 downto 3);
175
176         dpwm_reset <= resize((’0’ & mp0) + ("00" & unsigned(duty_in(13 downto 1))), 14);
177         mpt_passed <= ’1’;
178
179         counter_temp := (’0’ & counter_sig) + to_unsigned(1, 12);
180         counter_sig <= counter_temp(10 downto 0);
181
182       else
183         counter_temp := (’0’ & counter_sig) + to_unsigned(1, 12);
184         counter_sig <= counter_temp(10 downto 0);
185
186         for i in 2 to n_channels loop
187           dpwm_set_arr(i) <= resize((’0’ & mp0_arr(i)) +
188             (’0’ & not(’0’ & unsigned((duty_saved_arr(i)(13 downto 1))))) +
189             (to_unsigned(0, 14) & not((duty_saved_arr(i)(0))))  -
190             (unsigned(sync_in_arr(i)(2 downto 0))), 14);
191
192           if(counter_sig_arr(i) >= count_arr(i) or
193             (counter_sig_arr(i) = to_unsigned(0, 11))) then
194
195             counter_sig_arr(i) <= "00000000001";
196           else
197             counter_temp := (’0’ & counter_sig_arr(i)) + to_unsigned(1, 12);
198             counter_sig_arr(i) <= counter_temp(10 downto 0);
199           end if;
200         end loop;
201
```

```
202       end if;
203     end if;
204   end process;
205
206   process(clock, aclr) -- setd/clrd for ch1
207   begin
208     if(aclr = '0') then
209       clrd <= '1';
210       setd <= '0';
211       dpwm_reset_lsbs <= (others => '0');
212       dpwm_set_lsbs <= (others => '0');
213     elsif(rising_edge(clock)) then
214       if(counter_sig >= dpwm_reset(13 downto 3) and mpt_passed = '1') then
215         clrd <= (not(duty_one_arr(1)));
216         setd <= '0'; -----
217         dpwm_reset_lsbs <= std_logic_vector(dpwm_reset(2 downto 0));
218       elsif(counter_sig >= dpwm_set(13 downto 3) and mpt_passed = '0') then
219         setd <= (not(duty_zero_arr(1)) and not(sforce_arr(1)));
220         clrd <= '0';
221         dpwm_set_lsbs <= std_logic_vector(dpwm_set(2 downto 0));
222       elsif(counter_sig < dpwm_set(13 downto 3)) then
223         clrd <= (not(duty_one_arr(1)));
224         setd <= '0';
225         dpwm_reset_lsbs <= (others => '0');
226       else
227         setd <= '0';
228       end if;
229     end if;
230   end process;
231
232   process(clock, aclr) --mpt_passed for other channels
233   begin
234     if(aclr = '0') then
235       for i in 2 to n_channels loop
236         mpt_passed_arr(i) <= '0';
237       end loop;
238     elsif(rising_edge(clock)) then
239       for i in 2 to n_channels loop
240         if(counter_sig_arr(i) < mpt_arr(i)) then
241           mpt_passed_arr(i) <= '0';
242         else
243           mpt_passed_arr(i) <= '1';
244         end if;
245       end loop;
246     end if;
247   end process;
248
249   process(clock, aclr) -- setd/clrd for other channels
250   begin
251     if(aclr = '0') then
252       for i in 2 to n_channels loop
253         clrd_ch(i) <= '1';
254         setd_ch(i) <= '0';
255         reset_calctd_arr(i) <= '0';
256         dpwm_reset_arr(i) <= "00001100000000";
257       end loop;
258       dpwm_reset_lsbs_ch <= (others => '0');
259       dpwm_set_lsbs_ch <= (others => '0');
260     elsif(rising_edge(clock)) then
261       for i in 2 to n_channels loop
262         if((set_comp_arr(i) = '0')) then
263           clrd_ch(i) <= (not(duty_one_arr(i)));
264           setd_ch(i) <= '0';
265           dpwm_reset_lsbs_ch(3*i - 4 downto 3*i - 6) <= (others => '0');
266           reset_calctd_arr(i) <= '0';
267         elsif( mpt_passed_arr(i) = '0' and reset_calctd_arr(i) = '0') then
268           setd_ch(i) <= (not(duty_zero_arr(i)) and not(sforce_arr(i)));
269           clrd_ch(i) <= '0';
```

```
270           dpwm_set_lsbs_ch(3*i - 4 downto 3*i - 6) <=
271             std_logic_vector(dpwm_set_arr(i)(2 downto 0));
272
273           dpwm_reset_arr(i) <= resize(('0' & dpwm_set_arr(i)) +
274             ('0' & unsigned(duty_saved_arr(i))), 14);
275
276           reset_calctd_arr(i) <= '1';
277         elsif( rst_comp_arr(i) = '1' and mpt_passed_arr(i) = '1') then
278           clrd_ch(i) <= (not(duty_one_arr(i)));
279           setd_ch(i) <= '0';
280           dpwm_reset_lsbs_ch(3*i - 4 downto 3*i - 6) <=
281             std_logic_vector(dpwm_reset_arr(i)(2 downto 0));
282
283           reset_calctd_arr(i) <= '0';
284         else
285           setd_ch(i) <= '0';
286         end if;
287       end loop;
288     end if;
289   end process;
290
291   process(clock, aclr) -- input register. if number of channels is changed, this process
        must be updated appropriately.
292   begin
293     if(aclr = '0') then
294       freq_in <= "010100010100000";
295       duty_in <= "001001000110000";
296       duty_one_arr(1) <= '0';
297       duty_zero_arr(1) <= '0';
298       sforce_arr(1) <= '0';
299       for i in 2 to n_channels loop
300         duty_in_arr(i) <= "001001000110000";
301         sync_in_arr(i) <= "010100010010000";
302         duty_one_arr(i) <= '0';
303         duty_zero_arr(i) <= '0';
304         sforce_arr(i) <= '0';
305       end loop;
306     elsif(rising_edge(clock)) then
307       freq_in <= freq;
308       sforce_arr(1) <= sforce_ch1;
309       sforce_arr(2) <= sforce_ch2;
310       sforce_arr(3) <= sforce_ch3;
311       sforce_arr(4) <= sforce_ch4;
312
313       if(duty >= freq) then
314         duty_in <= freq;
315         duty_one_arr(1) <= '1';
316         duty_zero_arr(1) <= '0';
317       elsif(duty = 0) then
318         duty_in <= (others => '0');
319         duty_zero_arr(1) <= '1';
320         duty_one_arr(1) <= '0';
321       else
322         duty_in <= duty;
323         duty_one_arr(1) <= '0';
324         duty_zero_arr(1) <= '0';
325       end if;
326       -- the following must be changed if number of channels is changed.
327       if(sync_ch2 > freq) then
328         sync_in_arr(2) <= freq;
329       else
330         sync_in_arr(2) <= sync_ch2;
331       end if;
332       if(duty_ch2 >= freq) then
333         duty_in_arr(2) <= freq;
334         duty_one_arr(2) <= '1';
335         duty_zero_arr(2) <= '0';
336       elsif(duty_ch2 = 0) then
```

```
337         duty_in_arr(2) <= (others => '0');
338         duty_zero_arr(2) <= '1';
339         duty_one_arr(2) <= '0';
340       else
341         duty_in_arr(2) <= duty_ch2;
342         duty_one_arr(2) <= '0';
343         duty_zero_arr(2) <= '0';
344       end if;
345
346       if(sync_ch3 > freq) then
347         sync_in_arr(3) <= freq;
348       else
349         sync_in_arr(3) <= sync_ch3;
350       end if;
351       if(duty_ch3 >= freq) then
352         duty_in_arr(3) <= freq;
353         duty_one_arr(3) <= '1';
354         duty_zero_arr(3) <= '0';
355       elsif(duty_ch3 = 0) then
356         duty_in_arr(3) <= (others => '0');
357         duty_zero_arr(3) <= '1';
358         duty_one_arr(3) <= '0';
359       else
360         duty_in_arr(3) <= duty_ch3;
361         duty_one_arr(3) <= '0';
362         duty_zero_arr(3) <= '0';
363       end if;
364
365       if(sync_ch4 > freq) then
366         sync_in_arr(4) <= freq;
367       else
368         sync_in_arr(4) <= sync_ch4;
369       end if;
370       if(duty_ch4 >= freq) then
371         duty_in_arr(4) <= freq;
372         duty_one_arr(4) <= '1';
373         duty_zero_arr(4) <= '0';
374       elsif(duty_ch4 = 0) then
375         duty_in_arr(4) <= (others => '0');
376         duty_zero_arr(4) <= '1';
377         duty_one_arr(4) <= '0';
378       else
379         duty_in_arr(4) <= duty_ch4;
380         duty_one_arr(4) <= '0';
381         duty_zero_arr(4) <= '0';
382       end if;
383     end if;
384   end process;
385
386   process(clock, aclr) -- adc synchronization
387   begin
388     if(aclr = '0') then
389       adc_flag <= '0';
390     elsif(rising_edge(clock)) then
391       if counter_sig >= unsigned(adctime) then
392         adc_flag <= '1';
393       else
394         adc_flag <= '0';
395       end if;
396     end if;
397   end process;
398
399   process(clock, aclr) -- detecting glitch conditions
400   begin
401     if(aclr = '0') then
402       for i in 2 to n_channels loop
403         rst_comp_arr(i) <= '0';
404         set_comp_arr(i) <= '0';
```

```
405       end loop;
406     elsif(rising_edge(clock)) then
407       for i in 2 to n_channels loop
408         if (resize(('0' & counter_sig_arr(i)) + to_unsigned(1, 12) +
409         (to_unsigned(0, 11) & sync_glitch_arr(i)), 11) <
410           (dpwm_reset_arr(i)(13 downto 3))) then
411
412           rst_comp_arr(i) <= '0';
413         else
414           rst_comp_arr(i) <= '1';
415         end if;
416
417         if( (dpwm_set_arr(i)(13 downto 3)) > ((counter_sig_arr(i)(10 downto 0)) +
418           to_unsigned(1, 12)) ) then
419
420           set_comp_arr(i) <= '0';
421         else
422           set_comp_arr(i) <= '1';
423         end if;
424       end loop;
425     end if;
426   end process;
427 end rtl;
```

A.3 Output Multiplexer

```
-- This is an 8-input asynchronous digital multiplexer.
-- Written by Tero Kuusijarvi, 27.7.2017.

library ieee;
use ieee.std_logic_1164.all;

entity mux is
   port(
   sel : in std_logic_vector(2 downto 0);
   in0 : in std_logic;
   in1 : in std_logic;
   in2 : in std_logic;
   in3 : in std_logic;
   in4 : in std_logic;
   in5 : in std_logic;
   in6 : in std_logic;
   in7 : in std_logic;
   output : out std_logic);
end mux;

architecture rtl of mux is
begin
  with sel select
  output <= in0 when "000",
        in1 when "001",
        in2 when "010",
        in3 when "011",
        in4 when "100",
        in5 when "101",
        in6 when "110",
        in7 when "111",
        in0 when others;
end rtl;
```

A.4 Set-Reset Latch

```
-- This is an asynchronous SR latch.
-- Tero Kuusijarvi 23.2.2017

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity srlatch is
  port(
    s_in       : in  std_logic;
    r_in       : in  std_logic;
    q_out   : out std_logic );
end srlatch;

architecture rtl of srlatch is
signal temp_q, not_q : std_logic;
begin
      temp_q <= r_in nor not_q;
        not_q <= s_in nor temp_q;
    q_out <= temp_q;
end rtl;
```

B Floating-Point Arithmetic

In this appendix, the VHDL code for all of the implemented floating-point arithmetic modules are included as listings.

B.1 Number Format Conversions

```
-- This package contains floating point algorithms and conversions:
-- fp_to_unsigned32(op_a, result) -- Converts a normalized fp value to 32bit unsigned integer, expects normalized positive value.
-- unsigned32_to_fp(op_a, result) -- Converts a 32-bit unsigned integer to a floating point value.
-- Written by Tero Kuusijarvi, 15.3.2017.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package fp_package is
  procedure fp_to_unsigned32 (
    op_a  : in  std_logic_vector(31 downto 0);
    result : out unsigned(31 downto 0)
    );

  procedure unsigned32_to_fp (
    op_a  : in  unsigned(31 downto 0);
    result : out std_logic_vector(31 downto 0)
    );
end fp_package;

package body fp_package is
procedure fp_to_unsigned32 (
    op_a  : in  std_logic_vector(31 downto 0);
    result : out unsigned(31 downto 0)
    ) is
  variable temp_result_tot : std_logic_vector(31 downto 0);
  variable normalization_index : integer range 128 downto -128;
  begin
    temp_result_tot := (others => '0');
    normalization_index := to_integer(unsigned(op_a(30 downto 23))) - 127;
    if(normalization_index > 31) then -- overflow
      temp_result_tot := (others => '1');
    elsif(normalization_index > 23) then
      temp_result_tot(normalization_index) := '1';
      temp_result_tot(normalization_index - 1 downto normalization_index - 23) :=
        op_a(22 downto 0);

    elsif(normalization_index > 0) then
      temp_result_tot(normalization_index) := '1';
      temp_result_tot(normalization_index - 1 downto 0) :=
        op_a(22 downto 23 - normalization_index);

    elsif(normalization_index = 0) then
      temp_result_tot := std_logic_vector(to_unsigned(1, 32));
    else -- exponent is less than zero
      temp_result_tot := (others => '0');
    end if;
    result := unsigned(temp_result_tot);
end fp_to_unsigned32;

procedure unsigned32_to_fp (
    op_a  : in  unsigned(31 downto 0);
    result : out std_logic_vector(31 downto 0)
    ) is
  variable exponent_normalization : integer range -256 to 256;
```

```
57    variable temp_result : unsigned(23 downto 0);
58    variable temp_result_tot : unsigned(31 downto 0);
59    variable normalization_index : integer range 31 downto -1;
60    begin
61      temp_result_tot := (others => '0');
62      normalization_index := -1;
63      for y in 31 downto 0 loop
64        if(op_a(y) = '1' and normalization_index = -1) then
65          normalization_index := y;
66        end if;
67      end loop;
68      if(normalization_index > -1) then
69        temp_result_tot(30 downto 23) := (to_unsigned(127 + normalization_index, 8));
70        if(normalization_index > 22) then
71          temp_result_tot(22 downto 0) :=
72            op_a(normalization_index - 1 downto normalization_index - 23);
73        else
74          temp_result_tot(22 downto 23 - normalization_index) :=
75            op_a(normalization_index - 1 downto 0);
76        end if;
77        result := std_logic_vector(temp_result_tot);
78      else
79        result := std_logic_vector(temp_result_tot); -- all zeros
80      end if;
81  end unsigned32_to_fp;
82
83  end fp_package;
```

B.2 Subtraction

```
-- This is a floating-point subtractor module with 4 pipeline registers.
-- reset is an active-low reset.
-- op_a_in and op_b_in are IEEE 32-bit single precision floating-point
-- operands.
-- Written by Tero Kuusijarvi, 27.7.2017

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fp_subtract is
port(
    clk_in : std_logic;
    reset : in std_logic;
    op_a_in  : in  std_logic_vector(31 downto 0);
    op_b_in : in std_logic_vector(31 downto 0);
    result_out : out std_logic_vector(31 downto 0)
    );
end fp_subtract;

architecture str of fp_subtract is
signal op_a, op_b, op_a_reg1, op_a_reg2, op_a_reg3, op_b_reg1,
  op_b_reg2, op_b_reg3 : std_logic_vector(31 downto 0); -- for input registering
signal temp_result_tot_s, temp_result_reg1 : std_logic_vector(31 downto 0);

signal exponent_normalization, exponent_normalization_reg1 : integer range -256 to 256;
signal exponent_normalization_reg2,
  exponent_normalization_reg3 : integer range -256 to 256;

signal normalization_flag, normalization_flag_reg1 : std_logic;
signal shifted_signific : unsigned(22 downto 0);
signal normalization_index : integer range 22 downto -1;
begin
  process(clk_in, reset) -- input register
  begin
    if(reset = '0') then
      op_a <= (others => '0');
      op_b <= (others => '0');
      exponent_normalization <= 0;
    elsif(rising_edge(clk_in)) then
      op_a <= op_a_in;
      op_b <= op_b_in;
      exponent_normalization <= to_integer(signed('0' & op_a_in(30 downto 23)) -
        signed('0' & op_b_in(30 downto 23))); -- difference of exponents
    end if;
  end process;

  process(clk_in, reset) -- scaling the smaller operand
  begin
    if(reset = '0') then
      op_a_reg1 <= (others => '0');
      op_b_reg1 <= (others => '0');
      shifted_signific <= (others => '0');
      exponent_normalization_reg1 <= 0;
    elsif(rising_edge(clk_in)) then
      op_a_reg1 <= op_a;
      op_b_reg1 <= op_b;
      exponent_normalization_reg1 <= exponent_normalization;
      shifted_signific <= (others => '0');

      if(op_a(31) = op_b(31) and exponent_normalization > 0) then -- subtract
        shifted_signific <= not((shift_right('1' & (unsigned(op_b(22 downto 1))),
          exponent_normalization - 1)));

      elsif(op_a(31) = op_b(31) and exponent_normalization = 0) then -- subtract
        shifted_signific <= not((unsigned(op_b(22 downto 0))));
```

```
67      elsif(op_a(31) = op_b(31) and exponent_normalization < 0) then -- subtract
68        shifted_signific <= not(((shift_right(unsigned('1' & op_a(22 downto 1)),
69          (-exponent_normalization - 1)))));
70
71      elsif(exponent_normalization > 0) then -- add
72        shifted_signific <= shift_right('1' & unsigned((op_b(22 downto 1))),
73          exponent_normalization - 1);
74
75      elsif(exponent_normalization = 0) then -- add
76        shifted_signific <= (unsigned((op_b(22 downto 0))));
77      else -- add
78        shifted_signific <= shift_right(unsigned('1' & op_a(22 downto 1)),
79          (-exponent_normalization) - 1);
80      end if;
81    end if;
82  end process;
83
84
85  process(clk_in, reset) -- performing the subtraction
86  variable temp_result : unsigned(23 downto 0);
87  variable temp_result_tot : std_logic_vector(31 downto 0);
88  begin
89    if(reset = '0') then
90      exponent_normalization_reg2 <= 0;
91      temp_result_tot_s <= (others => '0');
92      op_a_reg2 <= (others => '0');
93      op_b_reg2 <= (others => '0');
94      normalization_flag <= '0';
95    elsif(rising_edge(clk_in)) then
96      op_a_reg2 <= op_a_reg1;
97      op_b_reg2 <= op_b_reg1;
98      exponent_normalization_reg2 <= exponent_normalization_reg1;
99      temp_result_tot := (others => '0');
100     temp_result := (others => '0');
101     if(op_a_reg1(31) = op_b_reg1(31) and exponent_normalization_reg1 > 0) then
102     -- subtract
103       temp_result_tot(31) := op_a_reg1(31);
104       temp_result(23 downto 6) := unsigned('0' & op_a_reg1(22 downto 6)) +
105         ('0' & (shifted_signific(22 downto 6))) + to_unsigned(1, 18);
106
107       normalization_flag <= not(temp_result(23)); -- subtract carry
108       temp_result_tot(30 downto 23) := std_logic_vector(op_a_reg1(30 downto 23));
109       temp_result_tot(22 downto 0) := std_logic_vector(temp_result(22 downto 0));
110     elsif (op_a_reg1(31) = op_b_reg1(31) and exponent_normalization_reg1 = 0) then
111       temp_result(23 downto 6) := unsigned('0' & op_a_reg1(22 downto 6)) +
112         ('0' & (shifted_signific(22 downto 6))) +  to_unsigned(1, 18);
113
114       normalization_flag <= '1';
115       temp_result_tot(31) := not(op_a_reg1(31) xor temp_result(23));
116       if(temp_result(23) = '0') then -- subtract carry
117           temp_result_tot(30 downto 23) := std_logic_vector(op_a_reg1(30 downto 23)) ;
118           temp_result_tot(22 downto 0) :=
119             std_logic_vector(not(temp_result(22 downto 0)) + to_unsigned(1, 23));
120       else -- no subtract carry
121         if(temp_result(22 downto 0) = to_unsigned(0, 23)) then
122           temp_result_tot(30 downto 0) := (others => '0');
123         else
124           temp_result_tot(30 downto 23) := std_logic_vector(op_a_reg1(30 downto 23));
125           temp_result_tot(22 downto 0) := std_logic_vector(temp_result(22 downto 0));
126         end if;
127       end if;
128     elsif (op_a_reg1(31) = op_b_reg1(31) and exponent_normalization_reg1 < 0) then
129     -- subtract
130       temp_result_tot(31) := not(op_b_reg1(31));
131       temp_result(23 downto 6) :=  ('0' & (shifted_signific(22 downto 6))) +
132         ('0' & (unsigned((op_b_reg1(22 downto 6))))) + to_unsigned(1, 18);
133
134       normalization_flag <= not(temp_result(23));
```

```
135         normalization_flag <= '1';
136         temp_result_tot(30 downto 23) := std_logic_vector(op_b_reg1(30 downto 23));
137         temp_result_tot(22 downto 0) := std_logic_vector(temp_result(22 downto 0));
138       elsif(exponent_normalization_reg1 > 0) then -- add
139         normalization_flag <= '0';
140         temp_result_tot(31) := op_a_reg1(31);
141         temp_result(23 downto 6) := unsigned('0' & op_a_reg1(22 downto 6)) +
142           ('0' & shifted_signific(22 downto 6));
143
144         if(unsigned(op_a_reg1(30 downto 23)) > to_unsigned(253,8)) then -- overflow
145           temp_result_tot(30 downto 23) := std_logic_vector(to_unsigned(254,8));
146           temp_result_tot(22 downto 0) := (op_a_reg1(22 downto 0));
147         elsif(temp_result(23) = '1') then -- add carry
148           temp_result_tot(30 downto 23) := std_logic_vector
149             (resize(unsigned('0' & op_a_reg1(30 downto 23)) + (to_unsigned(1,9)), 8));
150
151           temp_result_tot(22 downto 0) :=
152             '0' & std_logic_vector(temp_result(22 downto 1));
153
154         else -- no add carry
155           temp_result_tot(30 downto 23) := std_logic_vector(op_a_reg1(30 downto 23));
156           temp_result_tot(22 downto 0) := std_logic_vector(temp_result(22 downto 0));
157         end if;
158       elsif (exponent_normalization_reg1 = 0) then -- add
159         normalization_flag <= '0';
160         temp_result_tot(31) := op_a_reg1(31);
161         temp_result(23 downto 6) := unsigned('0' & op_a_reg1(22 downto 6)) +
162           ('0' & shifted_signific(22 downto 6));
163
164         if(unsigned(op_a_reg1(30 downto 23)) > to_unsigned(253,8)) then
165           temp_result_tot(30 downto 23) := std_logic_vector(to_unsigned(254,8));
166           temp_result_tot(22 downto 0) := (op_a_reg1(22 downto 0));
167         elsif(temp_result(23) = '1') then -- add carry
168           temp_result_tot(30 downto 23) := std_logic_vector(resize(
169             unsigned('0' & op_a_reg1(30 downto 23)) + (to_unsigned(1,9)), 8));
170
171           temp_result_tot(22 downto 0) :=
172             '1' & std_logic_vector(temp_result(22 downto 1));
173
174         else -- no add carry
175           temp_result_tot(30 downto 23) := std_logic_vector(resize(
176             unsigned('0' & op_a_reg1(30 downto 23)) + (to_unsigned(1,9)), 8));
177
178           temp_result_tot(22 downto 0) :=
179             '0' & std_logic_vector(temp_result(22 downto 1));
180         end if;
181       else -- add
182         normalization_flag <= '0';
183         temp_result_tot(31) := op_a_reg1(31);
184         temp_result(23 downto 6) := ('0' & shifted_signific(22 downto 6)) +
185           ('0' & unsigned((op_b_reg1(22 downto 6))));
186
187         if(unsigned(op_b_reg1(30 downto 23)) > to_unsigned(253,8)) then
188           temp_result_tot(30 downto 23) := std_logic_vector(to_unsigned(254,8));
189           temp_result_tot(22 downto 0) := (op_b_reg1(22 downto 0));
190         elsif(temp_result(23) = '1') then -- add carry
191           temp_result_tot(30 downto 23) := std_logic_vector(resize(
192             unsigned('0' & op_b_reg1(30 downto 23)) + to_unsigned(1, 9), 8));
193
194           temp_result_tot(22 downto 0):=
195             '0' & std_logic_vector(temp_result(22 downto 1));
196         else -- no add carry
197           temp_result_tot(30 downto 23) := std_logic_vector(op_b_reg1(30 downto 23));
198           temp_result_tot(22 downto 0) := std_logic_vector(temp_result(22 downto 0));
199         end if;
200       end if;
201       temp_result_tot_s <= temp_result_tot;
202     end if;
```

```
203   end process;
204
205   process(clk_in, reset) -- preparing for normalization
206   variable normalization_index_var : integer range 22 downto -1;
207   begin
208     if(reset = '0') then
209       op_a_reg3 <= (others => '0');
210       op_b_reg3 <= (others => '0');
211       normalization_flag_reg1 <= '0';
212       temp_result_reg1 <= (others => '0');
213       exponent_normalization_reg3 <= 0;
214       normalization_index <= -1;
215     elsif(rising_edge(clk_in)) then
216       normalization_flag_reg1 <= normalization_flag;
217       temp_result_reg1 <= temp_result_tot_s;
218       op_a_reg3 <= op_a_reg2 ;
219       op_b_reg3 <= op_b_reg2 ;
220       exponent_normalization_reg3 <= exponent_normalization_reg2;
221       normalization_index_var := -1;
222       for y in 22 downto 0 loop
223         if(temp_result_tot_s(y) = '1' and normalization_index_var = -1) then
224           normalization_index_var := y;
225         end if;
226       end loop;
227       normalization_index <= normalization_index_var;
228     end if;
229   end process;
230
231   process(clk_in, reset) -- output register
232   variable temp_result_tot_v : std_logic_vector(31 downto 0);
233   variable result : std_logic_vector(31 downto 0);
234   begin
235       if(reset = '0') then
236         result_out <= (others => '0');
237       elsif(rising_edge(clk_in)) then
238         temp_result_tot_v := (others => '0');
239         result := (others => '0');
240         temp_result_tot_v := temp_result_reg1;
241
242         if(normalization_index > -1 and normalization_flag_reg1 = '1' and
243           op_a_reg3(31) = op_b_reg3(31) and
244           (unsigned(temp_result_tot_v(30 downto 23)) > to_unsigned(23, 8)) ) then
245           -- final normalization
246           result(31 downto 23) := temp_result_tot_v(31) & std_logic_vector
247             (to_unsigned(to_integer(unsigned(temp_result_tot_v(30 downto 23)) -
248             (23 - normalization_index)), 8));
249
250           result(22 downto 0) := std_logic_vector(shift_left
251             (unsigned(temp_result_tot_v(22 downto 0)), 23 - normalization_index));
252         else
253           result := temp_result_tot_v;
254         end if;
255         result_out <= result;
256       end if;
257   end process;
258 end str;
```

B.3 Multiplication

```
-- This is a floating-point multiplier module with 2 pipeline registers.
-- reset is an active-low reset.
-- op_a_in and op_b_in are IEEE 32-bit single precision floating-point
-- operands.
-- multiply_enable must be set to '1' for operation.
-- Written by Tero Kuusijarvi, 27.7.2017

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fp_multiply is
port(
    clk_in : std_logic;
    reset : in std_logic;
    op_a_in  : in  std_logic_vector(31 downto 0);
    op_b_in : in std_logic_vector(31 downto 0);
    multiply_enable : in std_logic;
    result_out : out std_logic_vector(31 downto 0));
end fp_multiply;

architecture rtl of fp_multiply is
signal op_a, op_b : std_logic_vector(31 downto 0);
signal temp_result_a, temp_result : unsigned(47 downto 0);
signal temp_result_mult_a, temp_result_mult : unsigned(8 downto 0);
signal sign_b, sign_b_a : std_logic;
begin
  process(clk_in, reset) -- input register
  begin
    if(reset = '0') then
      op_a <= (others => '0');
      op_b <= (others => '0');
    elsif(rising_edge(clk_in)) then
      if(unsigned(op_a_in(30 downto 23)) = to_unsigned(0, 8)) then -- checking for zero
        op_a <= (others => '0');
        op_b <= (others => '0');
      elsif(unsigned(op_b_in(30 downto 23)) = to_unsigned(0, 8)) then
      -- checking for zero
        op_a <= (others => '0');
        op_b <= (others => '0');
      else
        op_a <= op_a_in;
        op_b <= op_b_in;
      end if;
    end if;
  end process;

  temp_result_a <= unsigned('1' & op_a(22 downto 6) & "000000") *
    unsigned('1' & op_b(22 downto 6) & "000000"); -- mantissa

  sign_b_a <= op_a(31) xor op_b(31); -- sign bit

  temp_result_mult_a <= ('0' & unsigned(op_a(30 downto 23))) +
    ('0' & unsigned(op_b(30 downto 23))); --exponent


  process(clk_in, reset) -- pipeline register
  begin
    if(reset = '0') then
      temp_result <= (others => '0');
      sign_b <= '0';
      temp_result_mult <= (others => '0');
    elsif(rising_edge(clk_in)) then
      temp_result <= temp_result_a;
      sign_b <= sign_b_a;
      temp_result_mult <= temp_result_mult_a;
```

```
    end if;
  end process;

  process(clk_in, reset) -- output
  variable temp_result_tot : unsigned(31 downto 0);
  variable vtemp_result_mult : unsigned(8 downto 0);
  begin
      if(reset = '0') then
        result_out <= (others => '0');
      elsif(rising_edge(clk_in)) then
        if(multiply_enable = '1') then
          vtemp_result_mult := (others => '0');
          temp_result_tot(31) := sign_b;
          if(temp_result_mult > to_unsigned(381,9)) then -- checking for overflow
            result_out <= std_logic_vector(temp_result_tot(31) &
              to_unsigned(253, 8) & not(to_unsigned(0, 23)));

          elsif(temp_result_mult < to_unsigned(127, 9)) then -- checking for underflow
            result_out <= std_logic_vector(temp_result_tot(31) & to_unsigned(0, 31));
          else
            vtemp_result_mult := '0' & resize(temp_result_mult +
              (('0' & not(to_unsigned(127, 8))) + to_unsigned(1, 9)), 8);

            if(temp_result(47) = '1') then
              temp_result_tot(30 downto 23) := (resize(vtemp_result_mult(7 downto 0) +
                to_unsigned(1, 9), 8));

              temp_result_tot(22 downto 0) := temp_result(46 downto 24);
            else
              temp_result_tot(30 downto 23) := vtemp_result_mult(7 downto 0);
              temp_result_tot(22 downto 0) := temp_result(45 downto 23);
            end if;
            result_out <= std_logic_vector(temp_result_tot);
          end if;
        end if;
      end if;
  end process;
end rtl;
```

B.4 Multiplicative Inverse

```
-- This is a floating-point multiplicative inverse module.
-- reset is an active-low reset signal.
-- op_a_in is a 32-bit single-precision floating-point operand
-- Wirtten by Tero Kuusijarvi, 27.7.2017

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY reciprocal IS
PORT(
    clk_in : std_logic;
    reset : in std_logic;
    op_a_in  : in  std_logic_vector(31 downto 0);
    result_out : out std_logic_vector(31 downto 0)
    );
END reciprocal;

ARCHITECTURE str OF reciprocal IS
signal op_a, op_b, op_a_reg1, op_a_reg2, op_a_reg3,
  op_b_reg1, op_b_reg2, op_b_reg3 : std_logic_vector(31 downto 0);

signal normalization_flag, normalization_flag_reg1 : std_logic;
signal shifted_signific : unsigned(22 downto 0);
signal normalization_index : integer range 22 downto -1;
begin
  process(clk_in, reset) -- Input register
  variable f32Denominator : std_logic_vector(31 downto 0);
  variable mult_inv : std_logic_vector(3 downto 0);
  type recipr_lookup is array (0 to 16) of signed(11 downto 0);
  constant recipr_points : recipr_lookup :=
    (x"7FF",
     x"70F",
     x"639",
     x"57B",
     x"4CF",
     x"432",
     x"3A3",
     x"322",
     x"2AC",
     x"23F",
     x"1DA",
     x"17C",
     x"125",
     x"0D3",
     x"08A",
     x"041",
     x"000" );

  type recipr_coeff is array (0 to 15) of integer range -30 to -8;
  constant coeffs : recipr_coeff :=
  (-30, -27, -24, -22, -20, -18, -16, -15, -14, -13, -12, -11, -10, -9, -9, -8);
  variable upperpoint, lowerpoint : signed(11 downto 0);
  variable coeff : integer range -30 to -8;
  variable linearapp : signed(15 downto 0);
  begin
    IF(reset = '0') THEN
      result_out <= (others => '0');
      f32Denominator := (others => '0');
    ELSIF(rising_edge(clk_in)) THEN
      f32Denominator := (others => '0');
      f32Denominator(31) := op_a_in(31);
      f32Denominator(30 downto 23) := std_logic_vector(to_unsigned(253, 8) -
        unsigned(op_a_in(30 downto 23)));

      mult_inv := op_a_in(22 downto 19);
```

```
67        upperpoint := recipr_points(to_integer(unsigned(mult_inv)));
68        coeff := coeffs(to_integer(unsigned(mult_inv)));
69        linearapp := (signed('0' & op_a_in(18 downto 12))*coeff);
70        f32Denominator(22 downto 10) := std_logic_vector((upperpoint & '0') +
71          (linearapp(12 downto 3)));
72
73        result_out(31 downto 23) <= f32Denominator(31 downto 23);
74        result_out(22 downto 12) <= f32Denominator(21 downto 11);
75      END IF;
76    end process;
77
78  end str;
```